Error Detection and Recovery in Software Development

Thesis

Error Detection and Recovery in Software Development

Tamara Lopez
B.S., Northern Arizona University
M.L.S., M.S., Indiana University

Centre for Research in Computing
The Open University

A thesis submitted for the degree of
Doctor of Philosophy in Computing

Submission date: 29th of February, 2016

Abstract

Software rarely works as intended when it is first written. Software engineering research has long been concerned with assessing why software fails and who is to blame, or why a piece of software is flawed and how to prevent such faults in the future. Errors are examined in the context of bugs, elements of source code that produce undesirable, unexpected and unintended deviations in behaviour. Though error is a prevalent, mature topic within software engineering, error detection and recovery are less well understood. This research uses rich qualitative methods to study error detection and recovery in professional software development practice.

It has considered conceptual representations of error in software engineering research and trade literature. Using ethnographic principles, it has gathered accounts given by professional developers in interviews and in video-recorded paired interaction. Developers performing a range of tasks were observed, and findings were compared to theories of human error formed in psychology and safety science.

Three empirical studies investigated error from the perspective of developers, reconstructing the view they hold when errors arise, to build a catalogue of active encounters with error in conceptual design, at the desk and after the fact. Analyses were structured to consider development holistically over time, rather than in terms of discrete tasks. By placing emphasis on "local rationality", analytical focus was redirected from outcomes toward factors that influence performance. The resultant observations are assembled in an account of error handling in software development as personal and situated (in time and the developer's environment), with implications for the changing nature of expertise.

Papers and Presentations

Portions of this research were presented at the following workshops.

Lopez, Tamara, Marian Petre and Bashar Nuseibeh. Active Error: Examining Error Detection and Recovery in Software Development. *Psychology of Programming Interest Group*, *Work-in-Progress Meeting, 2015*. School of Science and Technology, Middlesex University. London, UK, 8-9 January, 2015.

Lopez, Tamara, Marian Petre and Bashar Nuseibeh. Thrashing, Tolerating and Compromising in Software Development. *Psychology of Programming Interest Group*, *24th Annual Workshop, 2012.* London Metropolitan University, UK, 21-23 November, 2012.

Lopez, Tamara, Marian Petre and Bashar Nuseibeh. Getting at Ephemeral Flaws. *5th International Workshop on Cooperative and Human Aspects of Software Engineering, International Conference of Software Engineering*, 2012. Zurich, Switzerland, June 2nd, 2012.

To

Dr. Katharina Faust, Professor Vajkoczy
and to all the others
at the Department of Neurosurgery,
Charité Campus Virchow Klinikum, Berlin

Acknowledgements

First, I thank the managers who welcomed me, the stout-hearted developers who informed my studies and many who went before. At the turn of the century, I worked for companies that were “riding the wave” of the web. In that time I came to know men and women who were very good at writing code. They led me here, and I am so grateful.

My debts extend to Bashar Nuseibeh and Marian Petre who gave me the greatest gift of freedom to explore this topic. I could never thank them enough. There simply are no words to thank Marian for her support during the past two years. She never, ever let me go; others surely would have. I also humbly and deeply thank the faculty and staff members from the Open University who ensured I could complete this research.

Many folks in computing have permitted me to look through them, including Helen Sharp and Elizabeth Bjarnason, researchers at the Software Design and Collaboration Laboratory, U.C. Irvine and their industrial partners, and developers found on the internet. Though I suspect he is omniscient, Michael Jackson has always been gracious and encouraging. I likewise thank: Charles Hailey, Brian Randell, Thomas Green, Ben du Boulay, André van der Hoek, Alex Baker, Nick Mangano, Gerald Bortis, Thein Tun, Hugh Robinson, Yvonne Dittrich, Harold Osher and Lutz Prechelt.

In other places and times, a number of folks outside of computing taught me how to look. Mac, Buddy and Bob instilled in me the importance of play. Noriko Hara introduced me to Social Informatics, giving me the piece I had been looking for. Cesare Pastorino has long been my memex, may I never stop learning from him! Willard McCarty called me out for dithering six years ago, and set me firmly on this path. But I am so lucky that he first shared ideas and challenged me, as did: Jon Dunn, Dazhi Jiao, Stacy Kowalczyk, John Walsh, Bill Newman, Harold Short, Paul Vetch, Paul Spence, Arianna Ciula, Gerhard Brey and Vanda Broughton.

In these years, my husband has given me the finest example of steadfast pursuit. He is one heck of a guy. My girl and boy stood many mornings in doorways and drew their soft, soft hands from mine. They are the brave ones. I was so fortunate to find kindly other mothers to guide them, including Karen Harris, Sylvia Sieber and the *Erzieher/innen* at Outlaw Kita. I also give heartfelt thanks to *Signora* Maria Bella for elegantly guiding the ship and to Stella Chak, the greatest ever mother/sister/friend.

Finally, in the twenty years I spent getting to this moment, I travelled many miles away from my family in Arizona. Away, but not distant, I am thankful every day for the fullness of this surprising, blessed life. My life, given such a good start by each of you.

Table of Contents

Figures

Tables

1. Introduction

Determining what constitutes failure in software engineering is subjective and difficult to isolate. Boundaries between systems are fluid and the artefacts used to represent them complex. Perception and attitudes influence judgements about the causes of failure. The mechanisms designed to prevent failure are themselves failure-prone. The complexity of the topic has led different research communities to reinvent and rename related concepts. A tendency exists to overlook the ways in which various means of achieving dependability -and thus preventing failure- are relevant to one another (Randell, 1998).

By contrast, the concept of error in software engineering is stable, described using terms like ***fault***, ***defect*** or ***bug***. Bugs written into software produce undesirable deviations in specified behaviour (Avižienis, Laprie, & Randell, 2004). They must be tracked down so that they can be removed. It is not always possible to determine the circumstances under which a bug was written, or why. Nonetheless, they are widely considered to be the result of human error, attributed to poor understanding, inexperience, lack of skill, or incompetence.

This thesis considers a different sense of error. An ***error*** is also actively experienced, and may manifest only as a misunderstanding, or something that goes wrong and then is put right before a file is released, committed, or saved. Such errors are ephemeral, and as a result, there are often few material traces (Scott, 1990) left behind within code, descriptions or project records. The meaning associated with an error is personal. Its significance may diminish or develop over time as a developer takes on new projects, or faces new problems in different environments.

In the following pages of the thesis, the terms ***error*** as drawn from psychology and safety science refer to errors that are experienced. Other terms from these disciplines describe ***error handling***, the process by which developers detect, identify and recover from

errors. Particular instances of error handling are ***encounters*** or ***incidents***. The terms ***bug***, ***fault***, or ***defect*** signify error as conceived in software engineering.

The following section expands this conceptual foundation for human error. Next, a section presents the research question with working definitions and descriptions of influential factors. A brief statement highlights the qualitative analytical approach used in the reported studies. The chapter concludes with a guide to the text and a brief overview of each chapter.

1.1 Error Defined

Errors in the workplace are situational, particular; they arise in the form of "misfits" or "mismatches" between a person and a task or a person and a machine (Rasmussen, 1985, p. 5-6). Errors often unfold during normal, everyday actions (Norman, 2002). Things go wrong in the midst of "best attempts" to accomplish desired and reasonable goals (Lewis & Norman, 1986). They are encountered by workers at the "sharp end" (Woods, Johannesen, Cook, & Sarter, 1994)[1].

Errors arise, in part, because human performance is variable, marked by experimentation, by trial and error and "cutting corners" (Rasmussen, 1985). Variability is a natural and necessary part of learning and adaptation. It allows workers to be more efficient, to develop skills and improve performance. Erring is at times inevitable. Things may go wrong, but workers often are not at fault, given the demands of tasks and the conditions under which they perform (Hollnagel, 1998, p. 30).

The term "human error" is contentious. In the piece summarised above, Rasmussen prefers the terms ***misfits***, ***mismatches***, and ***malfunctions***, and argues that rather than human error, it may be more appropriate to identify features of "unkind" work environ-

1. The term "sharp end" is attributed here to Woods et al., but Woods attributes it to Reason, and other references suggest it originated with Rasmussen.

ments that cannot support variations in performance (Rasmussen, 1985). Hollnagel argues that the term "error" cannot be well defined and should instead be replaced with the terms ***action*** or ***activity*** (Hollnagel, 1983). In the context of medical safety, Woods writes that pursuing the question of what error is is a "dead end" (Woods & Cook, 2003, p. 2).

To declare that a human has committed errors that produce an accident or failure requires that judgements or causal attributions be made after the fact, based upon incomplete contextual knowledge about particular situations (Rasmussen, 1990). The boundaries of the error and its causes are determined in light of known negative outcomes. An analysis is thus blinkered by hindsight (Woods, Johannesen, Cook, & Sarter, 1994). Analysts select causes that are "familiar" (Rasmussen, Nixon & Warner, 1990), that can be measured using externalised criteria (Hollnagel & Amalberti, 2001).

In navigating a space of possibilities (Rasmussen et al., 1990), workers must redefine the goals and tasks they are given to perform. They transform them into individual plans and intentions for which actions can be undertaken (Frese and Zapf, 1994). Erring in the workplace is inevitable, and should be interpreted in the context of personal actions that are perceived to have been in error.

Error occurrences are ***actively*** experienced, they arise when planned sequence of mental or physical activities fail to achieve intended outcomes. (Reason, 1990, p. 9). A person becomes aware that he has made an error through feelings or perceptions that arise in the act (Sellen, 1994), based on suspicion or checks made of recently completed work (Allwood, 1984). He might also realise that an error has occurred by assessing "deleterious" outcomes (Norman, 1981). Recognition is made by comparing internalised aims, expectations and judgements (Rasmussen, 1985) to outcomes in the environment.

Error detection is a part of ***error handling*** (Brodbeck, Zapf, Prümper, & Frese, 1993): a person realises that something is wrong, identifies what went wrong and what should

have been done, and removes effects (Sellen, 1994). Handling an error may be more or less immediate (Norman, 1981) or may require drawn out, effortful problem solving (Reason, 1990).

1.2 Research Question

> "[A]lmost everyone who has ever written a program that did not immediately function as intended - a normal occurrence as we all know - has developed his personal theory about what went wrong in this specific case and why. " (Endres, 1975, p. 327)

The research reported in this thesis explores the common knowledge expressed by Endres in the quote above. It aims to understand more about the personal theories developers have about things that go wrong while making software, to catalogue specific instances, and to document the process employed to deal with them. To do this, it has addressed the following research question:

How do professional software developers detect, identify and recover from errors?

Though broad and intended to be exploratory, this question carries within it several related concerns:

Error: In software engineering, faults are discovered after software is written and, possibly, released. They are reported to developers as bugs. As defined in the previous section, errors arise from a personal action or actions that are perceived to be wrong. Individual experiences are the focus of inquiry. The research does not aim to establish causes but to explore the ***environmental and situational factors*** of occurrence.

Time: The meaning associated with errors is personal, and their significance may diminish or develop over time. This research collates data for analysis that gives a more realistic view of how time functions in software development. This point has two implications: it permits examination of how an individual error occurrence may ***transcend tasks and span time*** over the course of a project and reveals how perceptions toward errors ***change in response to the passage of time***.

Professional Developers: Empirical studies in software engineering often study performance in the laboratory or educational settings (Brandt, Guo, Lewenstein, Dontcheva, & Klemmer, 2009; Ko & Myers, 2005). Findings from cognitive task analysis suggest that the way people perform in the workplace is different (Crandall, Klein, & Hoffman, 2006). The analysis used data that depict professional developers and aimed to isolate examples of naturalistic performance.

Everyday Practice: The nature of the topic suggested that the examined practice would likely include challenging or even rare, "one-off" events. However, studies were not designed to perform a retrospective analysis of a large operational failure or another disastrous outcome. Instead, a commitment was made to examine ***routine, everyday practice***, with some limitations. For example, no study reviewed the process of agreeing to specifications with stakeholders. Likewise, reports of error were considered as developers utilise them, but not as users experienced and reported them.

Incidents: One aim of the research was to develop an understanding of software development by ***identifying incidents*** in everyday practice, not by examining particular tasks, methodologies or environments. An incident may have occurred within a particular task such as bug fixing or writing unit tests, but also during a design meeting, or in writing a method. Data were selected that included fine-grained detail about actions (Norman & Shallice, 1986) and performance (Rasmussen, 1985) that could be "tightly linked" to personal experience, to goals, to settings, and cues (Crandall et al., 2006, p. 21).

1.3 Approach

There is growing interest within the research community to find new ways to improve software quality by examining human error (Walia, Carver, & Bradshaw, 2015). This interest joins other, related calls that the research community must identify and articulate theory ((Ekstedt, Johnson, & Jacobson, 2012), and recognise that software engineering is a human activity (Captretz, 2014). Older examinations of human error in professional contexts performed retrospective analyses, using bug and modification reports, or retro-

spectively administered interviews and questionnaires that probe for detail about these two activities. A more general examination of errors that are made at other points is lacking.

To fill these gaps, this research has applied and evaluated a methodological framework for examining human error in software development. In addition to selecting appropriate methods for data collection, this research established analytical focus, determining what constitutes an error handling incident. This required identifying the boundaries of experience that relate to error encounters in software development.

The studies reported in Chapters 5, 6 and 7 were ***ethnographically-informed*** (Robinson, Segal & Sharp, 2007). The research collected data from interactions with and observation of developers in the field. A theoretical framework drawn from psychology and safety science was used to situate and interpret data.

It is not always easy to analytically establish the origins of errors (Hollnagel, 1983), nor for researchers to perceive what informants do (Geertz, 2000). Instead, this research considers software engineering as a human activity (Endres, 1975; Capretz, 2014) by examining how human error manifests in a socio-technical context (Rasmussen, 1990). The aim has been to understand what developers perceive with, the means by which or "through" (Geertz, 2000, p. 58) they handle errors encountered in daily work. Three dimensions were explored:

- The particular configurations of circumstances that provide material for problem-solving to developers (Reason, 1990).
- The process undertaken to detect, identify and recover from an error (Sellen, 1994).
- The feelings that influence process and resonate beyond problem occurrence (Reason, 1990).

1.4 Notes about the Text

This account has a kinship to the natural histories or framework studies described by Reason (1990). It is intended to provide a richer set of concepts for discussing error in professional software development. By extension, it stands to serve researchers in a number of fields that benefit from knowing what goes on during development practice.

To serve the broadest audience, the text is discursive, countering the skewed, inaccurate view of human error that can be conveyed by normalising human experience within typologies or models (Hollnagel & Amalberti, 2001). The approach is, instead, naturalistic (Le Coze, 2015), though findings in individual study Chapters 5, 6 and 7 and within the discussion given in Chapter 8 abstract individual experience into more general categories or themes that relate to established topics drawn from error handling research.

Errors are active, ***alive, teeming and writhing***, like the insects depicted in Figure 1.2. One aim has been to counter the image of error as specimens of insects that can be fixed with a pin and neatly ordered, as they are in Figure 1.1. This is a practice that has also been associated with older object-oriented or specimen focused cultural anthropology (Van Maanen, 2011).

Figure 1.1: Errors are specimens. Detail of "Butterflies", held by the Art and Picture Collection, The New York Public Library. Public domain.

Figure 1.2: Error is alive, teeming. Reprint of "Schutzeinrichtungen II", held by the Art and Picture Collection, The New York Public Library. Public domain.

1.4.1 Writing about People

The sense of error conveyed in this thesis was formed by studying the actions of a small number of people. Writing about people is hard, especially when it is necessary to write about their "less than perfect" aspects (Narayan, 2012, p. 46). The difficulties that arose in designing research on the topic of human error paled in significance to the subsequent difficulty of writing about observations. The goal in data collection was to minimise negative perceptions formed by colleagues and managers of developers who shared experiences. Reports had to be described respectfully while also acknowledging evidence of "clangers" or "dropped balls".

1.4.2 Writing about Error

We learn from mistakes, and errors are most clearly explained and understood through examples. Examples allow readers to associate the terms given to analytic categories with instances that are recognisable. They may be like or different from another person's experience, and compared with features of multiple researchers' data (Norman, 1981).

The term slip, for example, is relatively connotative, perhaps conjuring in your mind an image of a hapless encounter with a puddle of water or a banana peel. It takes on a meaning that is at once more precise and more general when given a definition like:

A slip is an action that was not intended or does not go according to plan.

A puddle of water or a banana peel may be involved, but may not be. Many plans and intentions are conceivable, after all, and many actions can be imagined to carry them out. "Slip" becomes a meaningful descriptive device when it is associated both with its definition and with examples that orient it to particular acts, such as speech:

> *"I was using a copying machine, and I was counting the pages. I found myself counting '1, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King.' (I have been playing cards recently.)" (Norman, 1981, p. 12)*

or physical activity:

> *"I caught myself as I was about to pour the tea into the opened can of tomatoes that was just next to (left of) the teacup. (The can was empty.)" (Norman, 1981, p. 12)*

In the following pages, the text depicts developers with pseudonyms and reports aspects of incidents using language that is realistic, "dispassionate" and in the third person (Van Maanen, 2011, p. 45). However, the text also makes extensive use of examples given by the developers. The aim is to represent their encounters in "experience-near" terms as they did: spontaneously, "un-selfconsciously", colloquially (Geertz, 2000, p. 57).

1.4.3 The Structure of the Thesis

The following chapters in the thesis are structured as follows.

- **Chapter 2** provides background literature for this examination of error, describing perspectives on the topic of human error in software engineering, and fields in psychology.
- **Chapter 3** establishes the commitment made to examine error handling in the context of actions taken within socio-technical environments.
- **Chapter 4** discusses how ethnographically-informed methods were used to conduct three studies.
- **Chapters 5**, **6** and **7** detail findings, which, taken together, identify features of error handling in software development.
- **Chapter 8** synthesises a view on error handling in software development.
- **Chapter 9** concludes the thesis with a review of contributions and implications for future research.

2. Background

The previous chapter established a refocused definition of error. Though an error may result in faults left behind within source-code (Avižienis, Laprie, & Randell, 2004), an error may also be actively detected, identified and recovered from in the course of practice. Error in this sense may leave no clear representation within software artefacts because effects are removed before software is released, or files are saved and committed to version control systems.

This chapter surveys perspectives on error in different disciplines, beginning with software engineering. The following section summarises literature from psychology and safety science that examines human action, error and aspects of performance in the workplace.

2.1 Error in Software Engineering

This section considers the topic of error in software engineering discourse. It starts with a description of the concept of dependability, followed by a brief overview of fault analyses. The third section, 2.1.3, is a critical review of root-cause studies, taken as an exemplar of fault analyses in industrial contexts.

This review was conducted by examining research and trade publications that treated concepts related to error and failure. Software engineers name things, giving operational definitions to concepts by specifying their attributes within typologies and models (Svenonius, 2000). Within the natural language used in the discourse, there are also conceptual definitions given, intentional and connotative statements that describe what is to be specified (Svenonius, 2000).

Both kinds of statements served as sources of evidence about how researchers and practitioners conceptualise error. The software engineering discourse employs literary rhetoric to serve different aims. Language is scientific; it enables "the audience to see the

world as it is" (Gusfield, 1976, p. 17). However, articles and papers are also cultural products (Kling, 1994); they use language in less neutral terms to persuade and communicate (Gusfield, 1976).

The topic of error was explored by performing detailed keyword searching of journals, by chasing citations within articles, surveys and roadmaps, state- of the art and -of the discipline pieces, and position papers. Journalistic sources and software engineering course work and syllabi that specifically address software failure were examined. Materials related to dependability were selected from work dating back to the 1960's with some representation from the '80s and '90s; the majority of materials examined are from the first decade of the 21st century.

2.1.1 Dependability

Since its identification during the 1960s as one of the key problems in computing (Buxton & Randell, 1970; Naur & Randell, 1969), the provision of reliable software and the prevention of large and small scale failure has been a core theme in software engineering research. These aims have been met, for example, by ensuring that software specifications are correct using mathematical proofs, or by designing and structuring systems to allow software to degrade gracefully in the presence of errors.

These and related areas of software engineering research make software more ***dependable***, a notion that is multivalent. Dependability has developed over several decades within software engineering research as an overarching concept that subsumes ***reliability*** and other attributes like availability, safety, confidentiality, integrity and maintainability (Avižienis, Laprie, & Randell, 2004, p. 5).

This overview takes the following form. The first section describes prior notions of ***reliability*** and ***correctness***, drawn from the reports of the 1968 and 1969 NATO confer-

ences. A second section explores a more recent concept, ***fitness,*** in the context of software and systems.

2.1.1.1 Reliability

The topic of reliability figured prominently at the 1968 NATO conference (Naur & Randell, 1969). Participants of the workshop noted a "conscious need" to consider reliability within the design process (Naur & Randell, 1969; p.44/26)[2]. They also linked reliability to the ability to achieve "freedom from mistakes" when software was in production (Naur & Randell, 1969; p.100/59).

Though the report stated a perceived need among software engineers to quantify notions of reliability, the problems associated with the issue were described in qualitative terms. User expectations were reported to be at once unrealistically high and low. Users were found to expect software to reach a state of "total system reliability". Even large systems were supposed to never, or rarely, fail over the course of decades. By contrast, other participants observed that user expectations were low, that customers were able to continue to work "even when everything is falling apart" (Naur & Randell, 1969, p.71/p.40).

Rising expectations for software performance were found to be a consequence of improvements in hardware. Expectations were said to be manageable through careful tolerance of errors in both software and hardware. Threats to meeting user expectations for reliability were associated with difficulties and costs associated with testing and integrating changes to software that was already in operation (Naur, 1969, p. 70/p. 40-41).

2.1.1.2 Correctness

At the time of the second NATO conference in 1969, the topic of ***quality*** subsumed the notion of reliability. Quality had two senses: ***correctness*** in performing specified tasks

2. The dual-page citations given for references to the NATO reports reflect changes in pagination in the accessed version of the reports. The first number signifies the page number in the original report. The second number reflects the location of the quote in the pdf report.

and efficient ***performance*** (Buxton & Randell, 1970). Correctness likewise was described in two contexts related to errors. ***Formal correctness*** concerned preventing errors from entering software, while ***debugging*** techniques centred on how to remove errors "when you have got them" (p. 20/p. 15).

At this conference, Dijkstra is reported to have made the famous comment that "[T]esting shows the presence, not the absence of bugs" (p. 20/p. 16). His position is of course well known (Dijkstra, 1972), but this review addresses this sentiment in the context of a paper he gave at the prior NATO conference, a year earlier.

Entitled "Complexity Controlled by Hierarchical Ordering of Function and Variability" (Naur & Randell, 1969), that paper anchored a discussion about the design process, and drew out details about the relationship between design and production. Dijkstra argued that the conviction of quality or "goodness" of software could not be achieved through testing, but instead must be proven before software is written. It would come out of the design process itself:

> *"If you have your production group, it must produce something, but the thing to be produced has to be correct, has to be good. However, I am convinced that the quality of the product can never be established afterwards. Whether the correctness of a piece of software can be guaranteed or not depends greatly on the structure of the thing made." (p.20/p. 16).*

The structure he references is the employment of a systematic method to produce software, a method which "gives proof" that the software is correct.

Other participants challenged Dijkstra's views. Counterpoints made by Willem van der Poel are summarised below. They are relevant to the research reported in this thesis because they address environmental and social aspects of development that influence notions of dependability, and by extension error handling. van der Poel argued that:

Errors can "cut right across" layers in systems and can have effects that are illogical, a phenomenon that has been described more recently as the "cause/effect chasm" (Eisenstadt, 1997).

It is necessary at times to "believe" in the correctness of a piece of software that was written by someone else.

Formulated precisely, the specification of a problem is equivalent to the solution. van der Poel asked not what is the solution, but *how do we solve the problem?* He questioned the "missing link" in Dijkstra's method, the elements of creativity, invention, intuition and process that could not be "symbolised or mechanised."

Some errors are due to the handling of software before execution. These comments regarded the material aspects of software of the time, of punch cards, transcriptions, of physical ordering and carrying to machines for execution (pp. 51-52/p. 30).

As Dijkstra noted in his rebuttal, van der Poel was interested in ***how*** programmers solve problems and in how they deal with problems that cannot be fully specified. These he addressed as matters of expertise: an unexplained process by which one comes to be gifted, to "know" how to manipulate formal specifications in practice. Dijkstra classed the other comments made by van der Poel together as representative of mechanical or clerical error: potentially costly if not dealt with, but manageable using available methods of the day.

2.1.1.3 Fitness

Recent writings draw dependability up out of the software toward its creators. The impression given is of dependability as an adaptive property that emerges and develops over time to meet the requirements of the environment and culture that create it. Dependability is formed amongst makers, who must confront a "thousand points of doubt" as they write software (Ferguson, 1992, p. p.183) and in so doing exhibit the senses of "fitness and adequacy" that characterise engineering practice more generally (Ferguson, 1992).

In the paper “How did software get so reliable without proof?” (1996), Hoare rejected the need for formal proof in all but the most critical components, instead favouring the establishment of a ”culture of reliability” enforced through code inspections, reviews and walkthroughs. He moved the notion of correctness away from design to development, noting that at the stage of fault removal, reliability in large systems may be a product of a ”natural symbiosis” achieved over many years of ***mutual adaptation*** between individual components maintained by humans.

A symbiotic, adaptive sense of dependability has also been used to explain the difference between the performance of systems and programs. Shaw contrasted the realities of ”real systems” with programs developed using methodologies of classical computer science (2002). Programs result in deterministic solutions to clearly defined problems. Systems, on the other hand, must respond even when the problem space is not completely understood, or requirements for behaviour change. They must be ***fit for purpose*** while maintaining the quality or ”health” of the system, however imprecisely defined.

Similarly, Randell’s work highlights the complexity of socio-technical “systems-of-systems”. Such systems have boundaries between them that are ”unknown and unknowable”. It is necessary to establish methodologies for achieving a more broadly conceived notion of dependability that can accommodate human values like trust (Randell, 1998).

In the context of faults and failures, Randell invoked Christopher Alexander’s thoughts to describe the nature of software design as being one of “fit”: the need to ascertain appropriateness in the current moment for an unknown future. Often, suitability is determined by identifying what ***does not work***, by detecting ***incongruities*** (Randell, 2003). Randell argued that in software engineering, such reasoning does not depend on specifications, but on personal, authoritative ***judgements*** taken during design and deployment.

Winograd and Flores characterise this sense of fitness as ***breakdowns*** (Winograd & Flores, 1987) and lend to it the factor of time. Breakdowns are detected by someone who has been broken out of readiness-to-hand, or "concernful" use of the tools and practices that underpin work. Errors are problems; they reveal inadequacies of commitment of "language in action". They are moments in which when someone's words have not had the intended effect on the computer. When a problem occurs, each person will bring a unique background, formed in the past, to understand how and why a problem occurred, whether the problem can be fixed or not, and the means required to fix it. (p.77).

2.1.2 Fault Analyses

The NATO conferences found that the problem in building quality software was "artisanal craft", while the solution was projected to be techniques and theories developed in software engineering (Buxton & Randell, 1970; Naur & Randell, 1969). One disciplinary branch of research that subsequently formed to address the problem of creating dependable software is fault analysis.

Fault analyses address the problem of quality by avoiding operational failure and employ a range of analytic methods including statistical analysis, program analysis, case study, formal methods and system analysis. With some exceptions (Magalhães, von Staa, & de Lucena, 2009), a particular study will often examine a single part of the development process, such as requirements engineering. The intention is to meet a single dependability aim, such as fault prevention. As their name suggests, ***fault prevention*** studies intend to prevent the introduction of faults during design and development. In addition to fault prevention, studies can be categorised in terms of other dependability aims: fault tolerance, fault removal, and fault forecasting (Pullum, 2001):

> ***Fault removal*** studies develop processes to remove faults written into software. As with prevention efforts, removal techniques cannot ensure that all faults are removed from a system because they can only determine whether or not software

matches the specified required behaviour. These analyses cannot determine that something was left unspecified.

Fault forecasting is likewise employed during software validation to indicate the presence of faults and to predict the risk of operational failures. It can be used to determine whether additional testing or other means should be applied to software before it is released.

Fault tolerance techniques enable systems to tolerate faults that are not removed before release. They do this by allowing operations to degrade gracefully and to recover from errors to prevent complete operational failure.

Studies across the categories, as summarised in Table 2.1 are empirical; they analyse existing bodies of software. However, they often employ quantitative, rather than qualitative analytical techniques. The intentional, connotative significance of concepts (Svenonius, 2000) related to software dependability are reduced into operational terms that can be measured and thus used to demonstrate, verify and validate that software meets a quantifiable, pre-determined degree of dependability. The lens of analysis is retrospective; examinations are not commonly made of software development practice as it occurs. All studies, however, are forward-looking, with general aims to improve software development process and practice in the future.

2.1.3 Root-causes

This section strengthens the case for examining errors in software as outcomes of human activity. It considers root-cause analysis studies to be both an exemplar of fault analysis research and a phenomenological source for identifying gaps in understanding about error in software development. The studies reviewed were selected by chasing footnotes and were published over the course of approximately twenty-seven years.

	Disciplinary Areas	Representative Studies
Fault prevention Prevent the introduction of faults during design and development	Requirements engineering Structured design Structured programming Formal methods Software reuse	Shaw, 2002 Than et al., 2009
Fault removal Remove faults during testing and verification	Software testing Formal inspection Formal design proofs	Hanebutte & Oman, 2005 Butler et al., 2010 Zou, 2003 Pugh, 2009 Briand et al., 2003 Cataldo et al., 2009
Fault forecasting Predict the occurrence of operational failures	Software testing Program analysis.	Bertolino & Strigini, 1998
Fault tolerance Enable systems to degrade gracefully	Error detection Diagnosis Containment and recovery	Sözer, Tekinerdoğan, & Akşit, 2009

Table 2.1: A summary of fault analysis research. This is a brief catalogue of analyses that treat smaller aspects of failure within systems. The catalogue is representative, not comprehensive.

Root-cause studies identify the kinds of faults that predominate in a system in order to determine how software engineering process can be altered to prevent fault occurrence. The studies draw data from bug and modification reports (Leszak, Perry & Stoll, 2002; Perry & Evangelist, 1985, 1987), but also make use of in-process questionnaires (Basili & Perricone, 1984) and retrospectively administered surveys (Perry & Stieg, 1993).

Data are analysed and classified into taxonomies that identify the root-causes for faults. The classified set of data forms the basis for additional examination of particular code features such as complexity (Schneidewind & Hoffman, 1979), interface defects (Perry &

Evangelist, 1985, 1987) or more generally, environmental factors that influence software dependability (Basili & Perricone, 1984).

2.1.3.1 Establishing a Model for Examining Root Causes

Albert Endres performed an early, influential root-cause analysis of software written for IBM in 1975 (Endres, 1975). The paper had two outcomes. Its principal outcome was the establishment of a root-cause taxonomy designed to plot fault distribution and frequency in systems programming, software characterised by the author as beginning with ”high quality” requirements that structurally degrade over time (p. 327). The second outcome was a meditation on the nature of errors in software programming, and reflection about how they should be studied.

The study examined a single release at IBM of the operating system DOS/VS. It drew data from failed test cases generated over a five-month period. Two sets of test cases were run: the first a series of regression tests to ensure that old functionality had not been compromised by new development and the second to simulate user inputs to the system.

The tests resulted in 740 faults. The original development team categorised the faults according to the protocol which should be followed to correct them. Four hundred thirty-two were deemed to be program faults – and thus not duplicates, documentation errors, hardware failures, operator errors, or feature requests. These formed the data for analysis. An analysis was performed to determine where, when, and why the fault was made who made it, what was done wrong, what would have prevented the fault and what would support detection.

The primary outcome of analysis was a taxonomy of distribution by type of error. This taxonomy included three main groups:

- faults related to problem understanding
- faults related to implementation

- mechanical errors such as spelling, or errors in integrating modules.

The study found that almost half of the 432 cases were could be attributed to programming technique, with suggestions given that better programming methodology would reduce the number. Notably, the remaining errors were found to be due to ***problem understanding***, a category that included communication, and knowledge of the broader ”possibilities and procedures for problem solving” (Endres, 1975, p.331). Endres attributed this finding to the complexity of the tasks, noting that the problems to be solved in systems programming are inherently ill-formed, dynamic, and require iterative changes. The functional demands of such systems, he argued, can only be properly understood when they are seen in use. To reduce faults in this class, Endres concluded that changes must be made to the development process, including the use of design and code walkthrough sessions, prototypes of functionality and user tests.

The study noted two significant limitations. It defined errors in the context of corrections made to source code. The number of errors equaled the number of failed test cases and did not consider other problems that might be found and corrected along the way, or those of which the programmer may have ”secretly been aware of for some time” (Endres, 1975, p. 330). The information provided in failed test case reports was sufficient to explain where and when an error was made, however, more information was required to determine who made the error. These data were gathered from conversations held with the development team.

2.1.3.2 Following the Model

Six root-cause studies that follow a research model like the one used by Endres’ are profiled and reviewed in the text that follows. See Table 2.2 for a summary of the studies.. As in Endres' case, the studies primarily examined data drawn from bug and modification reports filed by users and testers (Leszak et al., 2002; Perry & Evangelist, 1985, 1987).

Other studies drew data from in-process questionnaires (Basili & Perricone, 1984) and retrospectively administered surveys (Perry & Stieg, 1993). The study design in one case was experimental and examined purpose-built software (Schneidewind & Hoffmann, 1979). In the other studies, an empirical examination was made of software written for industrial environments, in a variety of languages and for different operating systems.

Taxonomies that represent the root causes of errors were the primary tool used in the analysis. Some schemes were theoretical, designed *a priori* by the researchers (Basili & Perricone, 1984; Schneidewind & Hoffmann, 1979) or developed in collaboration with members of the development team (Perry & Stieg, 1993). The taxonomy used in one study developed out of an analysis of error data (Perry & Evangelist, 1985, 1987). A second study used a scheme created earlier, adapting and extending it to represent additional information (Leszak et al., 2002). Developers were asked to classify errors using taxonomies supplied by researchers in two cases. Basili and Perricone included their classification in a change report form completed by programmers. Perry and Stieg surveyed programmers responsible for closing modification reports asking them to classify the error into one of nine fault type categories and to indicate the phase of testing in which the fault emerged.

Classified collections of faults provided a lens for examining other features of software such as complexity, interface defects or environmental factors that influence software dependability. Complexity was found both to correlate to error frequency (Schneidewind & Hoffmann, 1979) and not to (Basili & Perricone, 1984). Application programming interfaces were found to have particularly high frequencies of errors associated with them (Perry and Evangelist, 1985, 1987). These and other root causes were interpreted according to the costs of finding and fixing (Basili & Perricone, 1984; Leszak et al., 2002, Schneidewind & Hoffmann, 1979).

	Hypotheses/Aims	Characteristics of Data	Characteristics of Software
Schneidewind & Hoffmann (1979)	Hypothesis: Program structure has a significant effect on error making, detection, and correction. Aim: To find a complexity measure that can be used to guide program de-sign and resource allocation in debugging and testing.	173 errors 64 errors deemed to be potentially relevant to complexity of structure	Four projects undertaken by the same programmer Algol W for execution on the IBM360/67 Purpose-built code.
Basili & Perricone (1984)	Aim: To analyze the relationships between environmental factors and errors reported during software development and maintenance.	231 change report forms, created by programmers over a period of 33 months. Reports were verified by team manager, validated by research team; New development, but existing code re-purposed in some cases	Approximately 90,000 lines of code Primarily in Fortran for execution on an IBM 360 Aerospace (satellite planning studies).
Perry & Evangelist (1985, 1987)	Hypothesis: Interfaces are a source of problems in the development and evolution of large system software.	94 randomly selected modification reports submitted by testers 85 contained sufficient data for the study Software evolution.	350,000 non-commentary source lines C programming language Fault reports written against global header files. Domain unreported, researchers affiliated with Bell Labs and MCC.

Perry & Stieg (1993)	Aim: To determine general and application specific encountered during software evolution. Aim: To determine problems are found. Aim: To determine when problems are found.	Total sample size unreported. 68% of surveys were returned in each of two surveys Software evolution.	1,000,000 non-commentary source code lines, distributed real-time system written in C on UNIX Telecommunications (AT&T).
Leszak, Perry & Stoll (2002)	Aim: To analyze defect modification reports; establish root causes. Aim: To analyze customer-reported modification reports Aim: To propose improvement actions to reduce critical defects and to lower rework cost	427 Modification Reports representing 13 domains (func- tional units of software) New development (51%) and evolution.	900,000 non-commentary source code lines Language and environment unreported Telecommunications (Lucent)

Table 2.2: A summary of root-cause analyses. The research model established by Endres was also used in other root cause analyses. This table gives information for six such studies, highlighting study aims, characteristics of study design, and the environment under investigation.

The studies, like Endres' converge on one point: knowledge is one of the largest problems in software development (Perry and Stieg, 1993). However, the findings represent the notion of conceptual integrity in different ways. Endres described it as problems of understanding. The other studies conflate reasoning with constructs taken from software engineering. For example, Basili and Perricone found that roughly half of all errors related to requirement and functional specifications. Perry and Evangilist noted that 25 percent of the interface errors they studied were due to issues in design (Perry and Evangilist, 1987, Section 2 Background for the study).

The aim to go beyond the source code and to "get at" the reasoning process of developers in some cases prompted a second phase of data collection. Perry and Stieg designed a

survey for their case study that included a section for identifying the “underlying causes” of design and coding errors. Examples of categories included "Ambiguous design" and "Knowledge incomplete". All members of this category represented difficulties related to maintaining conceptual integrity (Perry & Stieg, 1993). Supporting Endres’ findings, their analysis indicated that lack of information dominated the underlying causes of the errors, while knowledge-intensive activities such as code inspections dominated the means of prevention.

Commentary about developer proficiency figures strongly, if indirectly in the studies. Schneidewind and Hoffman noted that their scheme was superior because it captured the ***flawed*** ”mental processes” of the programmer in representing ideas within source code (Schneidewind & Hoffman, 1979, p.282-283). Perry and Evangilist gave several causes for their error categories related to human performance, including several mentions of ***inexperience*** (Perry & Evangelist, 1987). Leszak et al. reported that a ***mismatch*** between the technical skills required and those available among workers is often the root cause of faults (2002). Echoing Hoare and the recommendations of Endres, Perry and Stieg concluded that process should be altered to include ”non-technological, people-intensive means of prevention” (Perry & Stieg, 1993).

In conclusion, on close reading the papers reveal that to fully understand why errors are made, information must be gathered about human understanding – where it is lacking, how it is coordinated and maintained (Leszak et al., 2002). The studies led by Perry and Leszak conclude with suggestions for follow-up work using methods to investigate the human element of errors, but only Endres’ discussed in any detail the generative qualities of error. As Endres argued, programming is a human activity shaped by an inner life of motivations and mental processes, of personal strategies developed to manage the work of programming. The sources of errors must, therefore, be considered not with regard to ***correction*** of

faults, but instead to ***intended implementations*** and subsequent outcomes (Endres, 1974, p. 329).

2.2 Human Error

Human error is an old and vast concern, far too immense to be comprehensively explored within doctoral research in computing. James Reason's *Human Error* has thus served both as an entry point to error concepts and literatures and as the foundation for understanding psychological concepts and theories.

Analysis of the literature began with Chapter 6, which surveys ***error detection and recovery*** research. Related ideas are woven through many chapters of the text, expressed in varying degrees of detail. ***Problem-solving*** performed during error handling is detailed in Chapter 3, within a presentation of the Generic Error Modelling Framework. The notion of ***active errors*** and their relation to intention is best described within Chapter 7, "Latent errors and systems analysis".

Perspective on performance in the workplace was developed using strands of research from safety science and organisational psychology. Rasmussen is possibly best known within software engineering for the ***skills-rule-knowledge*** framework of performance (Rasmussen, 1985) discussed in Section 2.2.3.2. However, in two pieces written in 1990, he firmly challenged the view that retrospective, causal analysis yields understanding about accidents in complex work environments. This represented a powerful shift in thinking within safety science. Following the argument he made forward, one finds an evolution in thinking about accident analysis, termed the "New Look" by Woods (2003), and the "Third Age" of safety by Hollnagel (2011), recently designated within resilience engineering (Hollnagel, Woods, & Leveson, 2006).

Working within organisational psychology, Michael Frese and Dieter Zapf situated examination of human error within office environments. Drawing upon the paper "Action

as the Core of Work Psychology: A German Approach" (Frese and Zapf, 1994) they described the tenets of goal-based action, characteristics of tasks that bridge personal intentions and work assignments, and developed a theoretic taxonomy of errors. In related work, they along with colleagues examined errors that arise in computer-based office work. This vein of research persists, utilised in a book published in 2011 that treated errors in organisations (Hofmann & Frese, 2011).

To assess the strength of the literature selected from these disciplines, citations patterns were compared and persistence of the ideas since 1990 was established. *Human Error* was published that year, and many sources and threads of analysis that were examined germinated in the years just before or just following that time. The three disciplines develop theoretical arguments using similar classes of psychological literature and often cite the same studies, such as Norman's "Categorization of Action Slips," from 1981. Rasmussen's work has been hugely influential beyond science safety (Le Coze, 2015), informing the work examined for this thesis of Norman, of Reason, of Frese and Zapf, and of cognitive task analysis (Crandall, Klein, & Hoffman, 2006).

Human error is often defined in relation to ***actions*** taken, described in the following in Section 2.2.1 alongside related concepts such as ***intention, attention***, and ***information and knowledge***. Additional sub-sections situate ***typologies of human error*** interpreted in relation to action and to performance.

2.2.1 Action Models

Actions are performed by identifying an intention, which is broken down into individual acts. The acts form a sequence that begins and ends as required to complete the action. While an action is underway, activity is monitored, and feedback is assessed to determine if intentions are being met. When an action deviates from an intention, an error has occurred (Norman, 1981).

This description of action comes from Norman's Action-Trigger-Sequence system (Norman, 1981), one of a cluster of models (Norman, 1981; Norman & Shallice, 1986; Reason, 1984) formulated to explain how "systematic" or "predictable" varieties of human error arise (Reason, 1990, p. 36). The models were developed by interpreting and comparing accounts of everyday activity.

For example, in categorising slips of action, Norman analysed a thousand incidents that included his own collection of accounts, and a compilation of similar incidents from other researchers. The incidents used by him in the analysis were recorded immediately after the occurrence, either by the person who made the error or by an observer. Reason's work with slips of action resulted in a behavioural classification of error categories, a theoretical action model, and a set of hypotheses about the cognitive mechanisms that fail when action slips occur (Reason, 1984). His data comprised 625 slips of action compiled out of catalogues developed in two studies. One study collected sixty-three diaries over seven days that included information about what happened when deviations in action were discovered and the completion of a set of standard questions that contextualised individual occurrences.

The Action-Trigger-Sequence system depicts action as a linear, horizontal sequence. It represents how people perform well-learned, habitual actions using pieces of stored knowledge stored that "direct the flow of control" of motor activity (Norman, 1981, p. 4). Norman is referring in this description to ***schema***, a term made familiar in computer science through the work of Minsky (Brewer, n.d.). Reason situates schema within psychology as higher-order, generic cognitive structures underlying all aspects of human knowledge and skill. Their workings are not consciously experienced, but they "lend structure" to perceptual experience and to the information that is stored or retrieved from memory (Reason, 1990, p. 35).

A person selects and activates an action schema when the current state matches the conditions under which it should be activated, but this is dependent on the perceived quality, or "goodness" of the match (Norman, 1981, p. 14). Actions may be initiated by environmental input, previously activated sequences or by thoughts, memories, and competing aims. Slips of action, or errors, occur because multiple sources of activation are possible and conditions are variable.

Intention

Intentions define actions. Without intention, there can be no selection of acts, no corresponding activity, and no assessment of completeness or correctness. Intentions are the result of "many considerations", including personal goals, decision-making and problem-solving (Norman, 1981, p. 5). Naturally, some errors arise in forming intentions. Norman's analysis of slips considered only errors, and by extension actions, for which an intention was stated. However, his scheme also represented errors of intent, such as performing a reasonable action in the wrong environment or forming the wrong intention because of incomplete information.

An intention has two components: the expression of the desired "end-state", and indications of how it is to be achieved (Reason, 1990, p. 5). Different actions require differently specified intentions. Small everyday actions become routine over time and do not require explicit specification. By contrast, a novel or ill-learned action requires greater intentional specificity until it too is repeated enough to become routine. In assessing activity, actions that did not meet prior intentions or were not properly executed are erroneous.

Stated again, actions are at times so well understood and familiar that they can be performed automatically (Norman & Shallice, 1986). They are ***routine, habitual***. They arise out of intentions that can be clearly stated and broken down into a series of physical

acts (Norman, 1981). Their familiarity "invokes" well-specified expectations (Sellen, 1994, p. 486).

Attention

Attention is paid to ensure that intentions are being met. This is done by comparing original intent —what one meant to do— with information or ***feedback***. Comparisons are made between information and **expectations**, that is what one expects to happen. (Norman, 1981; Reason, 1990). Information thus may come from internal sources, as in statements of intent or expectation, or external sources, as in the effects or outcomes that are produced when activities are undertaken.

Attention is variously described as leading to error, as preventing error, as necessary for diagnosis and for forming intention. Paying attention too closely to simple tasks can lead to errors, as can paying too little attention at key moments (Reason, 1984).

Conscious Control

Periodic attention is used to monitor routine tasks, however, it is not always sufficient. At other times attention is commanded, it must be "close and labored", so that consequences of actions can be assessed (Reason, 1984, p. 516). Activities that command attention are often ***novel***. This may be because they are not as well understood by the performer (Norman & Shallice, 1986), or arise out of "new" circumstances or unfamiliar sequences that generate unpredictability (Sellen, 1994, p. 486).

There are other special conditions in which "heightened awareness" or conscious control is required: when plans must be made, decisions taken or errors must be corrected. As noted, it is needed for tasks that are not well-learned or have novel sequences, but also for those deemed to be difficult or dangerous, or for actions that counter strong habitual responses (Norman & Shallice, 1986, pp. 2, 8).

Norman and Shallice accounted for conscious command within the Attention to Action model (1986). In this model, the supervisory attention system manages activity by drawing on multiples sources and types of information, including past and present states of the environment, of intentions, and awareness of prior actions and outcomes. It depends upon ***will***. Will must be exerted to meet intentions, even if it means performing actions that one does not want to do. The exertion of will requires attention, but also "conscious knowledge" of the particular end to be met. Norman and Shallice suggest that this knowledge must be formed ***before*** conscious control is exerted.

The model of human action given by Reason likewise gives emphasis to the force of ***needs*** in regulating action. Needs are the "motivational springs" of human action. In agreement with Norman and Shallice, Reason argued that attention or deliberate control must be exerted differently when intentions are in danger of not being met. For Reason, such moments are those in which intention assumes control as the "chief executive", responsible for organising plans, monitoring and guiding activity (Reason, 1984, p. 533).

The following sub-sections present typologies developed using theories of performance that are action-based. The first reiterates ***slips of action*** developed by Norman and Reason. The next sub-section describes Rasmussen's ***skill-rules-knowledge*** performance framework. Two models developed by Reason that combine slips of action with Rasmussen's levels of performance are described in Section 2.2.3.3. Finally, a model of action and error developed using Action Theory to describe organisational practice is summarised.

2.2.2 Slips of Action

Slips result from actions that do not go according to plan. An intention, aim or a plan might have been well-formed, but something goes wrong in performance. Slips may be overt or

covert, occurring during speech and motor action. They are shaped by intention, execution, and circumstance (Norman, 1981).

Reason described these errors as "trifling and usually inconsequential blunders" (Reason, 1984, p. 517). Slips of action can be caught in the act, just after the occurrence or after a long delay. Recovery might require several attempts, and some errors go completely undetected (Norman, 1981). However, once detected and identified, slips of action have a more or less obvious solution (Allwood, 1984). They take three forms: ***slips***, ***lapses*** and ***mistakes***.

To review, a ***slip*** results from an action that does not go according to plan or which was not intended (Norman, 1981). This kind of error is often observable as in slips of the tongue, of the pen, or in operation of a machine. However a slip but may only be apparent to the person who has slipped, as in a spoken sentence that is grammatically correct, but of incorrect significance.

Lapses are failures of memory that lead to a failed action. A person may forget a plan entirely or lose intention in the midst of performance (Sellen, 1994). Going to bed without taking medicine or wondering why one has entered a room are two examples. Lapses are often covert (Reason, 1990) and can only be detected by the individual who experiences them.

When a discrepancy arises between what one intends to do and what one expects to have happen, a ***mistake*** has occurred (Reason, 1990, p. 8). The intentions may have been inappropriate (Norman, 1981) or ill-formed (Norman, 1981; Sellen, 1994). The actions undertaken to meet an intention may have been correctly selected and correctly carried out, but the original intent was wrong. Specifying intentions for complex actions requires ***problem-solving***, a "blanket term" used by Reason to describe reasoning, judgement, diagnosis and decision making (1990, p. 158).

Reason associated ***mistakes*** with two kinds of problem-solving, ***tactical*** and ***strategic***. Mistakes occur when people make failures in "judging available information, setting objectives and deciding on the means to achieve them" (Reason, 1990, p. 54). Slips of action are usually detected by the person who slips using internal criteria. Correct performance can be tactically determined because the actions are simple, the intentions are well-formed, and solutions are clearly recognisable beforehand. By contrast, to meet the requirements of more complex intentions, correctness must be evaluated using external criteria. Success depends on two factors: correct goal definition and the ability to "recognise and correct deviations" from the path toward the end. Success or failure of strategic decisions can only be judged over time, in light of overarching or distant goals (Reason, 1990, p. 158).

2.2.3 Skills, Rules and Knowledge

The skill-rule-knowledge (SRK) framework models cognitive control of human behaviour. The framework can be used in analysis to explain errors in performance that arise during an emergency or within hazardous environments (Reason, 1990). The model was developed based on studies performed by Rasmussen using the think-aloud protocol (Rasmussen & Jensen, 1974). The framework has been used to examine errors made in writing HTML and CSS (Park, Saxena, Jagannath, Wiedenbeck, & Forte, 2013). Another notable study in software engineering by Huang, Liu, Song, & Keyal used Rasmussen's description of performance levels to interpret how differences in cognitive styles and personality types might influence the occurrence of coincident faults in software (2014).

The SRK was described by Rasmussen in multiple reports and articles, and is used and described by many of the authors surveyed for this research (Hofmann & Frese, 2011; Reason, 1990; Rizzo, Bagnara, & Visciola, 1987). One criticism of the model is that it presents a normalised view of human behaviour (Le Coze, 2015). In so doing, many

contextual details performance that should be represented by an analysis are omitted (Rasmussen 1985).

The explanation of the model used to guide research in this thesis was drawn from an invited talk given in 1984 titled *Human Error Data. Facts or Fiction?* (Rasmussen, 1985). This description of the model emphasises that performance on the job develops ***over time***. This emphasis is particularly relevant in the context of rule-based performance, which in later treatments (Reason, 1990) is described as the application by a worker of if-then logic.

Performance, as modelled in the framework, is controlled at three levels: skill, rule and knowledge. Each level represents the cognitive demand required to complete different tasks. Cognitive demand correlates to the degree of familiarity a worker has with an environment and the source and character of information that is used to adjust behaviour. Each level is briefly summarised in the following paragraphs, as well as in Table 2.3.

Skill-based or motor tasks require low-levels of control. The tasks are highly familiar, routine, and an individual adjusts behaviour in response to *signals* in the environment.

Rule-based or procedural activities are familiar, they are performed and controlled by past experience. In this case, know-how gained through individual or collective experience is applied within a situation as a "recipe". Rules form ***over time***, as similar situations are encountered to which a recipe applies. *Signs* that indicate the state of the environment or internal goals initiate or modify behaviour.

Knowledge-based tasks involve reasoning in unfamiliar situations or conditions. Goals are developed through analysis of a situation, and plans are physically and conceptually developed and tested. Information takes the form of *symbols*, meanings that an individual develops internally to explain the functional properties of the environment he is in.

Level of Control	Description	Goal	Situation	Information Source
Skill-based	Sensi-motor activities, performed without conscious control. They are "smooth", automated and highly integrated	Explicit	Familiar	**Signals** are indicators of the environment. They are temporal and spatial, with no inherent meaning
Rule-based	Procedural activities, developed through previous experience and others' "know-how"	Explicit or implicit, the situation suggests a particular convention	Familiar	**Signs** Activate or modify predetermined recipes. They refer to analogous situations or proper behaviour
Knowledge-based	Plan development and selection and testing, through trial and error or conceptually	Explicit, derived from analysis of a situation and guiding personal aims	Unfamiliar	**Symbols** defined by and in reference to internal understanding of the environment

Table 2.3: Rasmussen's skill-rules-knowledge framework (Rasmussen, 1985).

2.2.4 Generic Error Modelling Framework

Reason associated slips of action with Rasmussen's skill-rule-knowledge (SRK) performance behaviour framework within the Generic Error Modelling Framework (GEMS). GEMS has a history of use in software engineering research. Huang, Liu, and Huang mapped Reason's error modes to a taxonomy of common activities in requirement analysis and software development (Huang, Liu, & Huang, 2012). Ko and Myers likewise drew from the error typology developed for GEMS and respective modes of failure to define *cognitive breakdowns* in using programming environments. Breakdowns were associated with the concept of latent errors from the Swiss Cheese Model (described in the following

section, 2.2.5) and situated within a framework used to perform retrospective, causal analyses of human error in programming activity (Reason, Hollnagel, & Paries, 2006).

The Generic Modelling System is a context-independent framework for considering varieties of human error. It models the ways in which different kinds of performance error relate to one another, and the cognitive origins and sources of failure associated with each. At the heart of the GEMS model is a typology matching slips of action to the levels of performance in Rasmussen's SRK model (see Table 2.4 for a summary). Slips and lapses were related to skill-based performance. Mistakes were delineated into two types, one associated with rule-based and one with knowledge-based performance.

This delineation accounts for evidence suggesting that some kinds of mistakes fall *between* the categories of slips and mistakes. At times, when people slip or make a mistake, they select a behaviour from experience rather than assessing and responding to the situation at hand, a phenomenon Reason describes as *strong-but-wrong*. At other times, people exhibit failures in judgement, in forming and in determining how to meet intentions, all behaviours that have been associated with mistakes. Within the same incidents, these people also exhibit behaviour associated with slips, in that they favour strong-but-wrong practices.

Reason identified eight dimensions that distinguish error types, summarised in Table 2.3 below, and in the paragraphs that follow.

Skill-based activities are routine, non-problematic and carried out within familiar environments. As established in the models of action, slips and lapses occur due to failures of monitoring linked attention to activity. Rule- and knowledge-based activities are undertaken under less familiar circumstances, spurred by unexpected events. They are unplanned for, and call for deviation from the current plan (Reason, 1990, p. 56).

Rule- and knowledge-based performance centres around problem-solving that fluctuates between searches for rule-orientated solutions and conscious, effortful knowledge-based reasoning toward a solution. Mistakes are made in the course of activities due to a bounded, "keyhole" ability to view possible solutions or because of incomplete or inaccurate knowledge of the problem space (Reason, 1990, p. 167).

Type	**Activity**	**Attention/ Control**	**Detection, situational influences**	**Rate, Predictability, Expertise**
Slips and Lapses	Routine actions, changes in conditions assessed at the wrong time.	Attention misdirected, feedforward control	Easy, rapid, effective recovery; focus of attention, strength of assoc.	Abundant; predictable; novices lack routines, ability to abstract.
Rule-based Mistakes	Problem solving, changes anticipated but when and how unknown.	Conscious attention to task, feed-forward control	Difficult to detect, may require support; attention, strength of assoc., nature of task, training.	Abundant; predictable; novices lack routines, ability to abstract.
Knowledge-based Mistakes	Problem solving, changes are unanticipated	Conscious attention to task, feedback control	Difficult, may require support; task and circumstance	Few; harder to predict; mistakes by experts "look" like novice mistakes.

Table 2.4: Relating error types to performance. Adapted from (Reason, 1990) and (Hollnagel, 1998).

Performance of skill- and rule-based activities requires *feedforward control*, that is the selection and application of stored procedural and situational knowledge. By contrast, knowledge-based activities are controlled using *feedback.* Reasoning is functional in that a problem solver forms an internal mental model of the problem, and then sets local goals, for which actions can be selected and understood, observed and assessed for completeness

and success. This is an "error-driven," conscious process that is "slow, sequential, laborious" and constrained (Reason, 1990, p. 57).

Novices commit slips and rule-based mistakes due to a lack, or inappropriate selection of stored knowledge. Skilled, expert performance is distinguished by the presence and application of routines and rules for action that are formulated in more abstract ways than novices. However, in knowledge-based activities, even highly skilled workers will behave like novices when presented with a novel situation for which stored rules and routines do not apply.

2.2.5 Swiss Cheese Model

Reason's "Swiss Cheese" model represents how concealed, hidden failures and local triggering events combine during catastrophic events. It was designed to be a heuristic explanatory device, conveying that catastrophic accidents in organisations are generally caused not by a single error, but instead by the conjunction at a point in time of multiple, unlikely and unforeseen factors. It has subsequently been used as framework for retrospective accident investigation and as a measurement tool to assess the health or vitality of a system (Reason, Hollnagel, & Paries, 2006). Although the model has undergone several revisions, the description given here is drawn from the version presented in Chapter 7 of *Human Error.*

The model includes several components, depicted in related diagrams within *Human Error.* This summary highlights four: a general typology of production that can be taken to represent any industry working with complex technology, a typology of human weaknesses that contribute to accidents, a typology of kinds of human error, and a model depicting the dynamic process of accident occurrence. The production model and human weakness typologies are composed of five interrelated elements, summarised in Table 2.5.

Elements of Production	Contributory Human Weaknesses	State
Decision Makers *Set goals, strategies, allocate resources* *e.g.* Designers, architects, executive managers	Fallible resource allocations for safety. Due to: • Uncertain outcomes • Feedback often negative, intermittent • Poor safety is easy to blame on careless or incompetent operators	Latent
Line Management *Implement strategies* *e.g.* Operations, training, sales, maintenanc	Consequent (in part) to fallible decisions. • Poor training • Scheduling • Poor procedures	Latent
Preconditions *Infrastructural* *e.g.* Equipment, Personnel, schedules, codes of practice, environment	Consequent (in part) to mgmt. deficiencies. • Stress, Negative life events, • Imperfect awareness of a system, • Lack of motivation	Latent
Productive Activities *Synchronised Performance of humans and machines.*	Extrinsically defined in relation to particular hazards and situations. • Slips, Lapses • Mistakes • Violations	Active
Defences *Safeguards against natural or intrinsic hazards*	Personal safety equipment, physical barriers to hazardous material. • Redundancy, • Diversity, • Human and machine	Active & Latent

Table 2.5: Interrelations between production and human activities. Adapted from Reason, 1990, Figures 7.4, 7.5 & pp. 99-209. The productive activities element (highlighted) is the focus of analysis in this thesis.

As Table 2.5 above shows, weakness at one level of production is dependent on actions taken at a higher level and have consequences for production elements that follow. Within a system, the consequences may be latent or active. ***Latent errors*** may remain concealed in a system, with adverse consequences that become evident over time in combination with other factors. They are generally produced at levels of production that are removed in space and time from work at the "front line" (Reason, 1990, p. 173). ***Active errors*** occur on the front-line and have effects that are felt "almost immediately" (Reason, 1990, p. p.173).

The error typology used in this model groups human error into a typology of ***unsafe acts***. These are active errors committed by workers and include unintended actions such as slips or lapses, and also intended actions that are mistaken. ***Violations*** comprise a fourth category in this typology. Like mistakes, violations are committed with intent. They are deliberate deviations taken against regulated safety procedures (Reason, Manstead, Stradling, Baxter, & Campbell, 1990). Violations arise in the context of the social context of work, a context bounded by mores, rules, and procedures (Reason, 1990).

Accidents are unpredictable, they arise when an unsafe act is committed that correspond to at a point in time to breaches in safety defenses. This "unlikely" combination of events aligns along a "trajectory of opportunity" (Reason, 1990, p. 208), famously depicted within the diagram depicted in Figure 2.1 as multiple slices that correspond to different elements of production. In the version printed in *Human Error* on pp. 208*,* the slice representing "unsafe acts", was depicted with numerous holes placed between a layer of psychological conditions and layers of defence mechanisms. In later versions of the diagram, depicted below in Figure 2.1, slices with numerous holes were used to depict defences, barriers and safeguards in organisational settings.

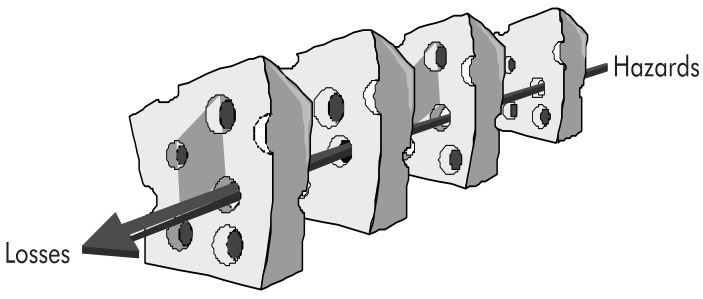


Figure 2.1: Reason's "Swiss Cheese" model. Reprinted from "Beyond the organisational accident: the need for "error wisdom" on the frontline." by J. Reason, *Quality and Safety in Health Care*, 13(suppl 2), ii28–ii33. Copyright 2004 BMJ Publishing Group Ltd. Reprinted with permission.

2.2.6 An Action-Oriented Taxonomy of Errors

In situating examination of errors within organisational psychology, a cluster of studies authored by researchers in Germany likewise adopted a conceptualisation of action that situates the concept of intention in terms of goals (Zapf, Maier, Rappensperger, & Irmer, 1994). Like Reason, Frese and Zapf acknowledge that in general, goals are preceded by needs, by "wishes" and "wants" (1994, p. 274), that translate into intentions that can guide action when an urgency or importance arises. However, Zapf and Frese mark a difference between personal actions and those taken at work.

Actions at work are linked to tasks, actions that must be performed according to rules in order to help meet organisational goals. In order to perform an organisational or external task, a worker must redefine it into internal tasks, and then to goals that can be met through action. The process of redefinition is described as one of interpretation, conducted based on professional and organisational knowledge, and prior experience.

The interplay between work tasks and personal goals influences aspects of the models of regulation and error. Hofmann and Frese present a recent synthesis of the German studies (2011), describing a four-level taxonomy of performance. Three of the levels correspond to those of the SRK, and by extension to slips of action.

> ***Skill-level* or *sensori-motor*** activities, as in the descriptions given by Norman and by Rasmussen, are those which are performed automatically, and which are monitored and adjusted based on feedback from the environment.
>
> ***Flexible action patterns*** are likened by Hofmann and Frese to schemata within Norman's action theory and rules in Rasmussen's SRK framework. The interpretation given to flexible action patterns in the German sense signifies a "ready-made" action sequence, that can be flexibly applied to meet *organisational* rules. It applies to situations in which a work task may require a set of established tasks that are routine, but not automatic. In this case, the rules are organisationally conceived and

followed, as in a set of documented procedures or checklists for performing maintenance tasks.

Conscious or intellectually regulated performance involves active reasoning. Goals must be considered, actions and sub-actions defined. It is undertaken in novel, unfamiliar situations. Action is conscious and effortful.

The fourth level is described as ***meta-cognitive***, describing *how* individuals formulate and undertake tasks to meet goals. This is a *heuristic* level of control that overarches action at all levels of conscious regulation. Heuristics guide how reasoning is performed: what kinds of plans are developed, which information search strategies are used, and how feedback from the environment is used. Heuristics are individual, and the Germans write that an individual may show a particular preference for a reasoning style, for example, always relying on their "gut" or by conducting a detailed search for information before taking action.

The interpretation of flexible action patterns is based on a narrow reading of both concepts. Schemata as used by Norman is only intended to represent how well-learned sensory or motor actions are stored in memory. He does not use this cognitive structure to explain how patterns of higher-level reasoning are cognitively managed.

Rasmussen's description from 1985 suggests that an individual may apply a "recipe" or a procedure to a situation, but the recipe has been developed through personal experience. The suggestion is given that the rule may be cultural, know-how that is provided to a person by a colleague, but it is not something that has been codified into a set of mandated procedures. It is not a rule that is followed, but rather one that is applied as in a "rule of thumb". The process of selecting the rule is described in terms of matching information from the state of the environment to memories of analogous situations.

The German researchers include several variations of an error taxonomy in their studies. The kinds of errors are correlated to the levels of regulation. In the most developed version of the taxonomy, movement errors accompany sensori-motor actions, while errors of habit, omission and recognition accompany flexible action patterns. They identify six error

variants at the conscious or intellectually regulated level of performance (Frese & Zapf, 1994; Hofmann & Frese, 2011).

Goal setting errors and ***thought*** errors relate to goal formation and execution. Goals may not be adequately developed or improperly decomposed into smaller goals. As noted by others (Sellen, 1994), the criteria for setting or assessing achievement may be vaguely specified. Thought errors occur when actions are "blinkered" and side effects and effects of time are not considered when plans are carried out.

Mapping errors relate to the collection, synthesis and actions taken upon information that is used in the course of action, while ***prognosis*** errors relate to the inability to adequately predict future system states.

Memory errors occur when a plan or part of a plan is forgotten in the midst of action.

Errors of ***judgement*** occur when a person does not understand or interpret information that is presented in the course of action.

2.3 Summary

Dependability is an old, multivalent concern in software engineering. A dependable service can be ***trusted***, but the trust must be justifiable. It must avoid failures that are more frequent and more severe than are acceptable to the user. Dependability is also assessed in terms of ***correctness***, an attribute that is gauged in relation to service and specification (Avižienis, Laprie, Randell & Jacquart, 2004). Correctness may be proven, but a system does not need to be correct to be dependable. It may also exhibit ***fitness***, an emergent, dynamic quality that develops in response to the needs of the environment and culture in which it is created.

Root-cause analysis studies improve software dependability by looking for the sources of faults in software. These studies use a simplified definition of error in order to produce measurable improvement. The simplification has limitations; it is difficult to adequately

explain why some errors occur, or to account for qualitative factors such as the effects of time and of human judgement.

At their simplest, actions can be performed automatically, with little or no attention paid to them. Actions that are well-learned or frequently performed form patterns that are stored in memory and can be re-used in the future. Actions that are simple or become routine may be performed with only periodic attentional checks. These checks ensure that intentions are being met by the actions that are being performed.

More complex intentions require that several actions unfold simultaneously and may require planning, analysis or decision making. By their nature, they require that conscious attention be paid to the tasks at hand. Such actions may also be novel, ill-learned, and the nature of the intention may preclude full understanding beforehand of outcomes. The acts taken to meet complex intentions are performed consciously, by paying "close and labored" attention (Reason, 1984, p. 516).

Error is a "generic term" encompassing occasions when planned sequences of mental or physical activities fail to achieve intended outcomes. Errors do not arise by chance, people commit them (Reason, 1990, p. 9). They may manifest at low levels, as in physical actions, or at higher levels, as in ***mistakes*** made in problem-solving (Norman, 1981; Reason, 1990). Error detection and recovery are more difficult in high-level problem-solving than in motor or skill-based activities because the process is subjective, it relies on goals that have been set for an undetermined future (Reason, 1990).

Error occurrences are often ***ephemeral***, they are imperfectly represented in the world after recovery. This type of error is experienced, and must be managed using intrinsic and extrinsic sources of information. Conditions are likely ***novel***, new or new again. As a consequence, the experience of managing an error is immediate and immersive, pulling one away from routine performance and directing attention to the particular action at hand.

Errors are sometimes “caught” in the act, but they may also be recognised after a delay in time.

In everyday error, the human is engaged in an action when something goes wrong, spurring an error handling process. In software development research, error handling is often described as being part of a managed process, triggered by a separate outcome-based detection and reporting process. Empirical studies of software engineering that examine aspects of bug fixing or maintenance, for example, generally describe the process as one of developers beginning from a reported outcome of faulty behaviour, working to establish a root cause for the error, and then determining how best to fix it (Ko & Myers, 2005).

The next chapter argues that human errors are a natural consequence of performance on the job (Rasmussen, 1990). They should be examined in terms of actions rather than of causes.

3. From Establishing Causes to Examining Actions

Front-line operators, managers and designers commit errors. Sometimes these errors result in critical failure. Moving from forming these conclusions to making suggestions for improvement is difficult (Rasmussen, Nixon, & Warner, 1990). The analysis, performed retrospectively, depends upon causal explanation and a correspondingly narrow definition of human error.

Causal analyses must establish a chain of significant events “upstream” from a negative outcome. The establishment of events depends on a subjective determination of stop-rules, pragmatically defined by analysts to determine how far back in time analysis must go. Conditions will therefore be explained by "abnormal, but familiar" events and acts, and causes will tend to reflect concerns relevant to a discipline at the time the analysis is made. Causal analysis assumes that the sequence in which an error is analysed can be “taken for granted” (Rasmussen, 1990, p. 1186).

3.1 Operational Failure in Software Engineering

Operational failure in software engineering is often examined in terms of systems-of-systems, complex environments with boundaries that are difficult to distinguish. (Randell, 1998). The aim of analysis is to identify weak elements within organisations, operations and software. As in other branches of engineering (Levy, Salvadori, & Woest, 2002), these studies are retrospective, performed after a service outage as a way to understand what went wrong, and who was responsible.

In general, operational analyses examine ***sudden*** and ***progressive*** failures of software, though this should be treated as a soft categorisation. Systems which primarily exhibit characteristics of progressive failure could suddenly fail, and sudden failures may show evidence of progressive issues when analysed.

Sudden failure is service outage on a large scale, often involving a critical piece of software. Individual or multiple faults become active at a moment in time or within a clearly bounded interval of time, and result in a large, catastrophic or spectacular system failure. Sudden failures have been examined in the context of medical devices (Leveson & Turner, 1993), aero-space engineering (Nuseibeh, 1997), and energy services (Than, Jackson, Laney, Nuseibeh, & Yu, 2009)

Progressive Failure arises in software systems that are deemed "good enough" to be released into production but which include significant problems that require maintenance, redesign and redevelopment, or that result in overextended resource allocation. Often this software is conceived and implemented within an already failing or flawed organisational or system initiative. Recent studies include examinations of medical transport scheduling (Dalcher & Tully, 2002) clinical records (Randell, 2007), and social services case management (Ince, 2010).

The case studies produced by these analyses often do not conclude with specific, precise reasons for failure, instead offering identifications of the system or sub-system that failed, and general recommendations for improvement going forward. Even when studies do isolate weaknesses in the processes of software creation or in particular software components, they do not tend to produce general frameworks or models that can be extended to improve software engineering practice.

Commentary about operational failure within grey literature is influential in shaping discussion about computing, and the directions that computing research takes (Kling, 1994). It is found in unpublished workshop and conference presentations (Easterbrook, 2005), within course work materials (Dix, 2003), and in journalism (Barker, 2007; Bogdanich, 2010; Charette, 2005; Garfinkel, 2005). In many aspects, these sources conform to the genre identified by Kling: they universalise technological experience, can

take extreme value positions, and describe technology as a dominant force in social interactions (1994). Popular treatments are often strongly anti-utopian, while workshop and conference presentations make claims that are more moderate. However, both present cases simply and draw on spectacular examples of failure.

Retrospective analyses are powerful, they use stark imagery that is compelling and easy to understand. However, a retrospective lens cannot provide insight to the internal, subjective criteria that may direct action (Hollnagel, 1998). It is distorted by the same weaknesses in human cognition that have been found to contribute to error, including perceptual biases and *strong-but-wrong* belief. We as arbiters know how things turned out. The people working at the sharp end did not, could not (Reason, 1990).

3.2 A Space of Possibilities

A "naturalistic" view on human error (Le Coze, 2015) better represents how error arises in modern work environments. Modern working conditions are socio-technical, and therefore different from earlier work environments. Workers operate within a dynamic space of possibilities (shown in Figure 3.1) and they must employ different skills to operate technology that is not stable and to meet ill-defined goals. Successful completion of tasks requires constant exploration of and an interaction between personal resources, accepted ways of doing things and resources for accomplishing them (Rasmussen, 1990).

Rasmussen describes the navigation process as one of adaptation and learning. Task completion depends upon continuous exploration, the development of strategies for decision making, and active control over selecting the path toward goals (Rasmussen, 1990). Errors in such environments are often not critical, they are every day, likely to arise during routine activity (Reason, 1984). They are an inevitable side effect of the process of exploration, acceptable and expected to be a natural consequence of of testing and crossing

the boundaries of knowledge, of resources and values within an organisational environment (depicted in Figure 3.2).

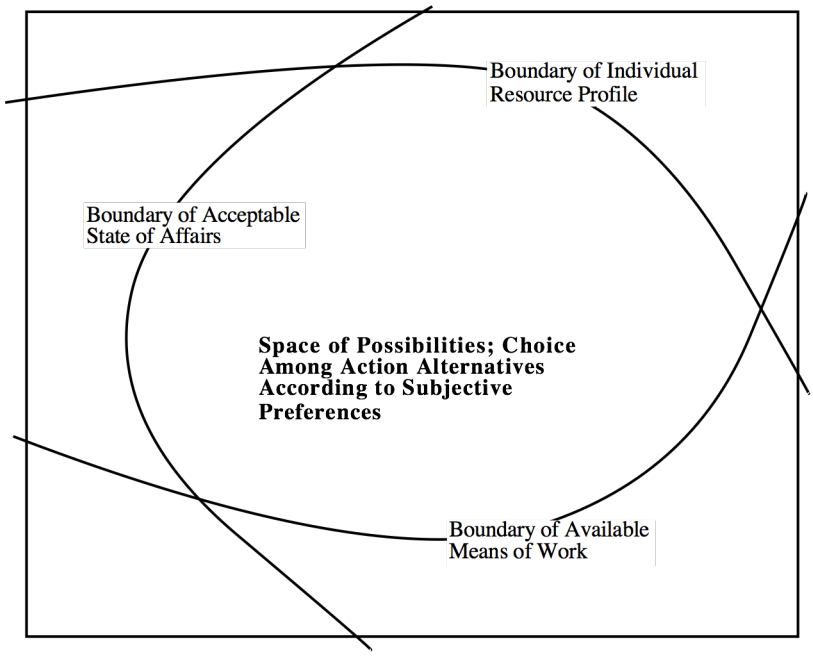


Figure 3.1: Rasmussen's space of possibilities. Reprinted from "The role of error in organizing behaviour." by J. Rasmussen, *Ergonomics*, *33*(10-11), p. 1191. Copyright 1990 by Taylor & Francis. Reprinted with permission.

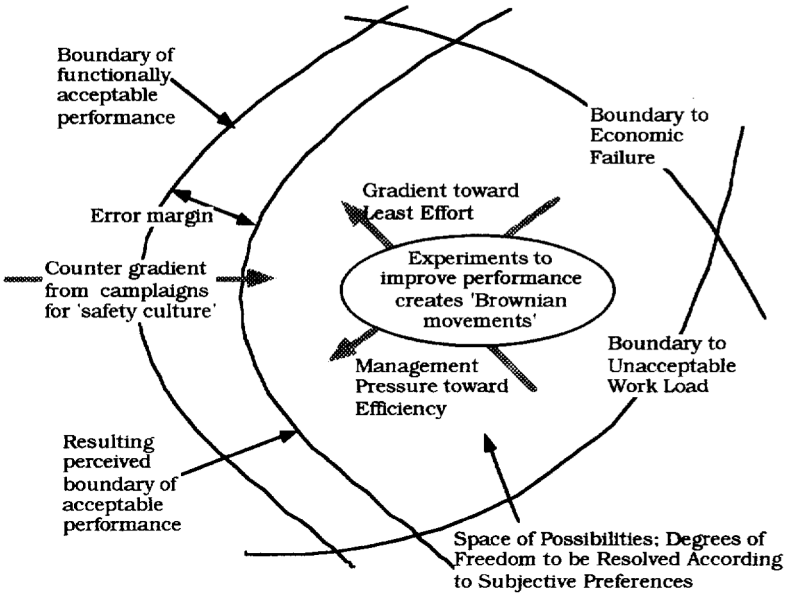


Figure 3.2: Rasmussen's boundaries of acceptable performance. Reprinted from "Risk management in a dynamic society: a modelling problem." by J. Rasmussen, *Safety Science*, *27*(2), p. 190. Copyright 1997 Elsevier. Reprinted with permission.

Taking a socio-technical view, it is not possible to establish a causal trace that has been "deflected from its intended course toward one goal" (Rasmussen, 1990, p. 1186). Instead, events are fluid: several goals and side effects unfold at once, resources are not stable, performance depends upon workers who have been granted and are exercising freedom of choice.

3.2.1 Actions

The notion of the space of possibilities is at the heart of the "third wave" (Hollnagel, 2011) of safety science, a framework for error that models natural or "ecological" safety (Amalberti, 2001, p. 117). Naturalistic examination of error need not establish blame for accidents, but strive instead to understand how contributory factors of individual and organisational activity produce safety. The framework assumes that mistakes are "cognitively useful" flags in the process of learning and impossible to eliminate. In terms of assessing performance, understanding how errors are detected and recovered from is more important than examining failures in production.

Ecological safety is achieved by the ways in which individual workers maintain awareness of the situation in which they are performing. Awareness is informed in relation to action: assessment and knowledge of possible actions, knowledge of difficulty, application of attention, choices made about how and whether to avoid error, error handling mechanisms, and the tolerance and recognition that some errors will still occur.

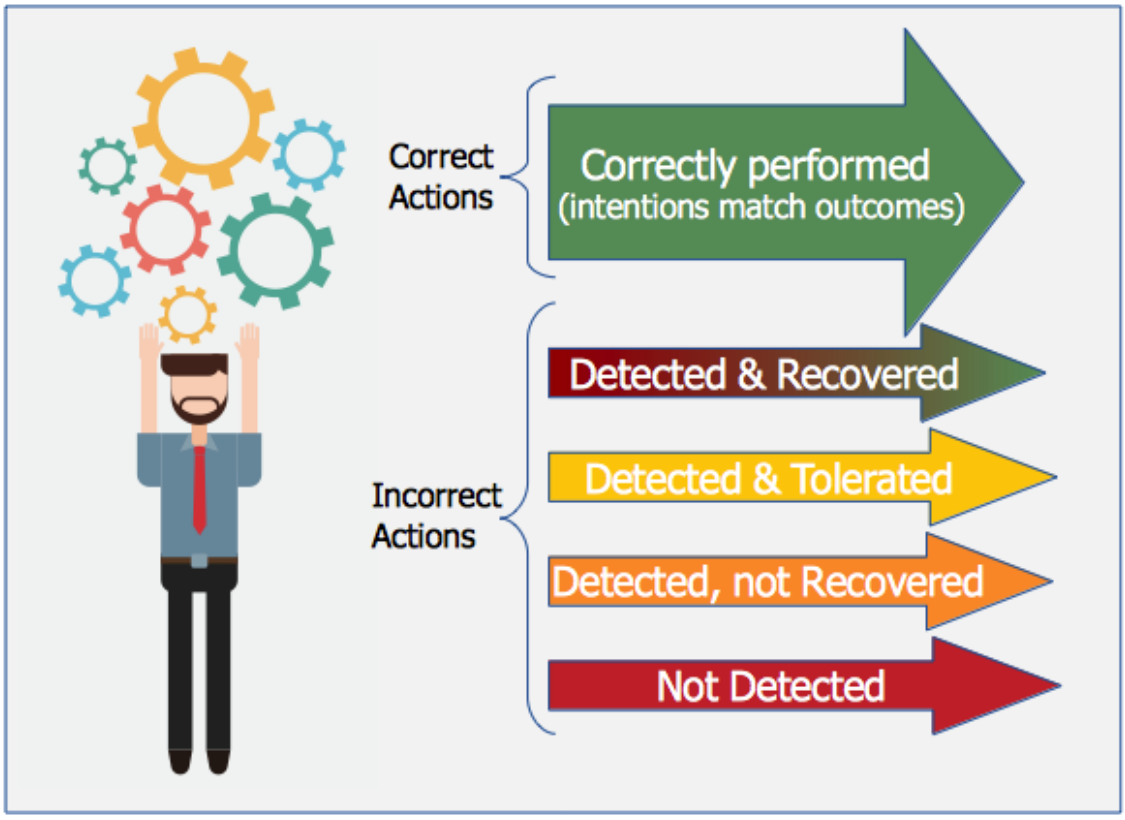


Figure 3.3: Actions and intention. Adapted from "The emperor's new clothes: Or whatever happened to "human error"" by E. Hollnagel and R. Amalberti. (2001). Proceedings of the 4th international workshop on human error, safety and systems development (pp. 1-18). Adapted with permission (Vector design: Vecteezy.com).

This view challenges the notion of a binary distinction between right and wrong actions, or between correct and incorrect (Hollnagel & Amalberti, 2001). Instead, actions as they relate to intention can be considered, as depicted in Figure 3.3, above.

This model demonstrates that operational failure is only one possible outcome of an error. Errors may also be handled: recovered from, tolerated, identified or missed. Analysis of incidents can be used to examine circumstances surrounding error occurrence, the perceptions of the worker in relation to that occurrence, and the significance of the error to a worker's broader working life.

3.3 Error Detection and Recovery

To handle an error, a person must know that an error has occurred, identify both what was "done wrong" and "what should have been done" and then understand how to "undo" the

effects of the error (Sellen, 1994, p. 476). Studies that have specifically examined aspects of error detection and recovery in psychology are surveyed below. The following section highlights the methods used in four groups of studies to collect and analyse data, followed by a synthesis of findings that relate to detection, identification and recovery.

3.3.1 Related Research

Surveys of error detection research already exist. James Reason surveyed the literature in Chapter 6 of *Human Error*, providing a general description of the error detection process that remains relevant, along with a description and analysis of key studies. Erik Hollnagel treated error detection studies in Chapters two and three of *Cognitive Reliability and Error Analysis Method* (Hollnagel, 1998). His purpose was to draw together psychology and safety science to articulate an updated model of human error, and to highlight existing approaches to the topic relevant to safety science. Most recently, David Hofmann and Michael Frese surveyed the literature in the introduction to *Errors in Organizations* (2011). An overview of representative papers is given in Table 3.1.

As described in the previous chapter, the studies of Reason and Norman examined collections of slips of action that were self-reported using a form of diary collection. Utilising Norman's slip classification, Sellen examined a collection of 600 self-reported errors collected using diaries for modes of detection (1994). Allwood used think-aloud protocol to examine how students detected and recovered from errors while solving set statistical problems (Allwood, 1984).

Studies conducted by Rizzo, Bagnara and Visciola examined the process of error detection in the use of computer software in office and steelwork settings. The studies used think-aloud technique and drew upon the evaluation process identified by Allwood, and the GEMS framework developed by Reason. Their participants performed set tasks using computer software, or simulated tasks in the steelworks.

In a series of studies, Brodbeck, Zapf, Frese et al. developed an error taxonomy developed out of action theory that they validated through observation of office workers using computer software (Dieter Zapf, Brodbeck, Frese, Peters, & Prümper, 1992). Their field study relied on constrained access to office workers in several German companies who performed a range of tasks in computerised environments.

3.3.1.1 British/North American

In the studies reported by Sellen, Norman and Reason, errors are represented in association with particular actions. Self-reports were recorded as soon as possible after occurrence, either by the person who erred or by an observer. Norman's reports included information about what the person was thinking and how the slip was discovered (Norman, 1981). Sellen and Reason used diaries kept by participants that recorded details of the error. Reason asked participants specifically to record information about intention, while Sellen asked respondents to describe circumstances surrounding how the error was detected and identified.

Findings are descriptive, intended to provide a framework for discussing error (Reason, 1990), presented within typologies (Norman, 1981) or more loose categorisations (Sellen, 1994). Norman developed a typology of action slips that describes the behaviours exhibited when the error occurred, that is whether a person made omissions, insertions, substitutions or reversals (Reason, 1984, p. 530). Sellen used a modified version of Norman's taxonomy as an analytic. Her own findings about detection were categorised in terms of details of monitoring and feedback that accompany erroneous acts. When error data is used to describe behaviour more generally, studies may include models such as the models of action developed by Norman and Reason (Norman, 1981; Norman & Shallice, 1986; Reason, 1984).

3.3.1.2 Swedish

Errors can also be defined in terms of the problem-solving process that is undertaken after detection. Allwood's research identified activities related to error based on data collected through think-aloud protocol. In his study, students enrolled in statistics classes were asked to perform set tasks with pre-determined correct solutions that related to their coursework.

3.3.1.3 Italian

Error handling has also been examined in terms of tasks performed in work settings. These studies examined the relationship between the three action-based error types defined by Reason in the GEMS model, and the three self-monitoring detection processes identified by Allwood. Sixteen subjects undertook three training sessions, followed by four experimental sessions in which the subjects had to perform increasingly complex tasks with a database system. 924 errors were made, 780 were detected. The study found that most skill-based slips were detected during error-hypothesis episodes. Most knowledge-based mistakes were detected based on suspicion.

Studies	Method	Error Taxonomy	Error Handling
British North American	Diary Study self-reports	Slips Lapses Mistakes	Action-based, Outcome-based, Cued by the environment
Swedish	Think-aloud set problems	Solution method High level math Other types Skip errors	Direct-error hypothesis; Standard Check; Error Suspicion
Italian	Think-aloud Computer set tasks Industrial simulation	Skill-based • slips or lapses Rule-based • mistakes Knowledge-based • mistakes	Direct-error hypothesis Standard Check Suspicion

German	Time-slice observation Interview Questionnaire	Skill-based • movement Flexible-Action Pattern • habit • omission • recognition Conscious • goal formation • mapping • prognosis • thought, memory, judgement	Internal • goal comparison • planning barrier External • evident information • system message • limiting function)

Table 3.1: An Overview of Error Detection and Recovery Research. Several of the studies had different combinations of collaborators. To ease reporting, and following Reason's convention of grouping the studies by nation (Reason, 1990), the papers are grouped by linguistic or geographic contexts: British-North American studies, Swedish, Italians and Germans.

Findings in the first Italian study were supported in a second study in which steel workers were asked to carry out a simulated production planning exercise. The exercise was recorded and analysed. 95 errors were made, 74 were detected. Taken together, the studies demonstrated a consistent association between error type and detection mode. Slips are detected most often by error hypothesis, while rule-based errors are detected using a combination of hypothesis and suspicion. Knowledge errors are largely discovered through standard checks employed in the course of work.

3.3.1.4 German

The German researchers interviewed, observed and administered questionnaires to office workers at eighteen organisations. Observation of work at computers were made that lasted in the range of 90 minutes. During this time, observers sat at the desk with the worker, and identified errors that occurred. These were classified according to the error taxonomy outlined in Section 2.2.3.4. Typing errors were not recorded. The error handling process

was considered to have begun at the point of detection, handling time was estimated from this point to the point of recovery. The observers also noted that external support materials (including consultation with colleagues) were used, distractions and emotional responses to the process (Brodbeck, Zapf, Prümper, & Frese, 1993).

3.3.2 Detection

Errors can be detected in the midst of action, on the basis of outcomes, or environmental cues (Reason, 1990). Detection occurs during "evaluative" problem-solving, at breaks during which previous actions following a standard practice or a spontaneous check that arises out of "perceived discrepancies" between actions and expectations for results (Allwood, 1984, p. 414).

Detection is independent from understanding the nature and source of the error (Zapf & Reason, 1994). An error detection process may be undertaken by a person regardless of whether or not an error is actually present. Likewise, error handling may not result in a clear identification and elimination of an error (Allwood, 1984, p. 414).

3.3.2.1 Action-based detection

In action-based detection, it is the act itself that provides information that an error has occurred, not the effects of the act. Errors are detected through perception of the act, not on perception of effects that the action has on the world. Reason describes actions ***not-as-planned*** as resulting from the failure of high-level attention. The tasks are usually automatically performed, but require occasional checks to make sure that intentions are honoured, particularly when they involve "deviation from routine practice" (Reason, 1990, p. p.157). Sellen found that this mechanism presents in two cases, summarised below.

A mismatch may arise between the stated action plan and executed actions. In these cases, the guiding intention might be sound, but the execution is "misordered or inaccurately articulated" (Sellen, 1994). Errors of ordering, blending or wrong insertions may be

given. These kinds of errors might be evident to others, such as when someone makes a spoonerism or blends words together (Norman, 1981), however Sellen postulates that it is internal perception that triggers detection.

3.3.2.2 Outcome-based detection

Error detection can occur after an action is completed. In this case, detection is related to expectations for outcomes, to the effects of outcomes or to anticipation that an error will occur. Intentions, and the corresponding expectations may be well or vaguely specified. Sometimes this is due to familiarity. If a situation is new, it may be unpredictable, while if it is familiar it will have well-specified expectations. Sometimes it is due to complexity. Tasks that require problem solving, such as mathematics, often don't have well-specified intentions.

Detection may occur as a result of mismatches between conscious intentions and executed actions. In this case, detection involves an evaluation process that begins with identifying the action (I did this), and examining the intentions (I meant to do this). It is outcome-based. Following Norman, Sellen describes the monitoring process in terms of information, asking in her analysis what information served as the basis for detection of an error, and with what was the information compared (1994).

Information could come from properties of the action itself, outcomes of the action or, properties of the environment in which the action was undertaken that prohibit or in some way alter completion of the act. She writes less explicitly about what that information is compared to, but in reading her categorisations, it is clear that the information is compared with expectations for outcomes, or with an evaluation of intention.

Detection arises when ***mismatche***s are perceived between the information upon which an error is detected and the comparisons made between that information and the "criterion or reference against which that information is compared" (Sellen, 1994, p. p.480).

Outcome-based detection may also occur as a part of ***standard checks*** (Allwood, 1984) undertaken to "look out" for familiar error patterns. Sellen described this in the context of writing dates after the turn of a new year. In these cases, people become sensitive to the possibility that an error might occur, and adapt their behaviour to check for it. Allwood described this as occurring independently, not out of a sense or perception that the solution was strange.

3.3.2.3 Environmental Conditions

Environmental information influences error handling. Physical constraints and cues spur error detection. System responses and feedback from colleagues is used to assess and set goals to support identification and recovery.

The environment may constrain action, thereby triggering an error detection. Norman defined these as ***forcing functions***, Sellen as "limiting functions". In this case, the action is physically barred, and cannot be completed. Forcing functions are unambiguous, and error detection is guaranteed (Reason, 1990). Reason somewhat ambiguously states that forcing functions may be a natural part of a task as in locks and bolts or deliberately designed into the environment as in computer software.

Sellen provides a better example of these two states, describing them as barring or subtle. In her example, trying to into the system Sellen also suggests that the environment might also provide a more "subtle" constraint, as when a person tries to unzip a button-fly. The second category is like Norman's mode errors, in which the action is not appropriate for the environment.

In the case of ***cued discovery***, failures arise in considering the larger problem space in which an action takes place. Reason describes cues not as constraints but rather as moments in which the environment provides opportunities "for rejoining the correct

path" (p.163). Here Reason describes following a "blinkered line of thought from one difficulty to the next" while attempting to change a tire on a car.

Information in the environment is commonly ***evident*** (Frese & Zapf, 1994), it arises, for example, from interactions with computer software. The environment provides information to users in the form of system responses (Lewis & Norman, 1986) and receives information from users based on actions taken. By contrast, actions that must be internally assessed will result in system responses that are not evidently wrong. Lewis and Norman identified ways in which systems "should" respond to erroneous actions: by gagging, providing warnings, doing nothing, self-correcting, engaging in dialogue, and by asking the user to "teach" the system what was intended.

An error may be brought to attention by someone else and reported (Reason, 1990), cases that are not covered in great detail in the error detection literature. Slips of the tongue are reported as being noted by an observer (Norman, 1981). Reason reported that Woods' examination of nuclear power plant operators found that "fresh eyes" were necessary to detect errors in diagnosis in complex situations, while operators were able to self-detect most slips (Reason, 1990). In office work, errors detected by colleague or clients, were reported as occurring infrequently in the midst of work (Frese & Zapf, 1994).

3.3.3 Identification and Recovery

Determining what should have been done may be clear from the circumstances or by the nature of the task, but may require more investigation, particularly if a situation or task are complex or unfamiliar (Sellen, 1994).

The action models of Norman and Reason are concerned with simple actions that can often be performed more or less automatically. Intentions and expectations are generally well-formed in these cases and the assessment of information is likewise swift. These models do not directly address handling for complex actions, how error identification is

achieved, or how a recovery is made. To establish a basis for error identification and recovery as it may unfold while performing more complex actions, it is necessary to draw on a broader discussion of problem solving.

3.3.3.1 Progressive and Evaluative Problem Solving

Allwood found that problem solving proceeds in two phases: ***progressive problem solving***, in which subject tries to achieve a goal, and ***evaluative***, in which the subject reviews completed work (1984). During an evaluation, a person might be satisfied with previous activities. Error handling is assumed to happen during negative evaluation episodes, when a person is not satisfied with previous activities. Allwood identified three types of negative evaluations: standard checks, direct-error hypothesis formation and suspicion of error. Allwood's analysis also represents other activities within error suspicion episodes. For example, diagnoses were identified, as were hypotheses and detection. Qualitative aspects of suspicion were also represented, including expressions of discontent and giving up.

3.3.3.2 Tactics and Strategies

Problem solving is both tactical and strategic (Reason, 1990). It requires defining goals, and then using tactics to forge an "adequate path" to achieving them. The process hinges on discovery, it may involve "inspired guesses" but also "trial and error". The criteria for success or failure is often only revealed with the benefit of hindsight. Intentions carry with them expectations, and are thus prone to confirmation bias - there is less objective information on which to base strategic decisions, while subjective influences restrict the search for cues that indicate that a choice was wrong or "inadequate" (Reason, 1990).

Reason situates identification and recovery in terms of *problem configuration*. The configuration is made of a set of cues, indicators, signs, symptoms and calling conditions. The set is immediately available, and are used in the handling process (Reason, 1990, p.

87). They demand different approaches and tactics. He identifies three kinds of configurations:

Static: In this case, the problem is fixed and independent of activities undertaken by a problem solver. They may be represented as abstract or concrete. (Examples: syllogism; Watson card test; cannibals-and-missionaries)

Reactive-dynamic: Changes in the problem space are a result of actions taken by the problem solver. Problems in this configuration may be direct, in which effects are immediately apparent, or indirect, in which the solver relies on augmented sensory aids for feedback. (Examples: jigsaw puzzles; assembly tasks; the Tower of Hanoi problem)

Multiple-dynamic: The configuration may change in response to actions taken by the problem solver, and also due to independent factors in the system or situation. Problems may be bounded, in which the independent variability is constrained and known, or complex, in which variability has multiple sources which are unconstrained and unpredictable. (Examples: bounded-chess; complex-nuclear power plant or medical emergencies)

3.3.3.3 Problem Solving and Performance

Skilled performance requires constant switching between skill, rule and knowledge based activity. When activities are routine and familiar, they proceed in a largely automatic fashion, with periodic checks to ensure that intentions are being met. Within the GEMS model, if a check identifies a threat to meeting intentions, rule-based problem solving will take place. If a rule is found that matches conditions, it will be enacted and activity will return to skilled performance. If no rule-based solutions are found, the model suggests that an effortful, conscious, knowledge-based problem solving process will be undertaken.

The initiation of knowledge-based reasoning does not preclude ongoing searches for patterns out of the rule repertoire. Rapid switching can occur between knowledge- and rule- based activities in order to form and execute a recovery plan in ***local problem***

solving, a process of establishing local goals that can be carried out and assessed. In this case, routines or rules will be borrowed from other established activities.

3.3.3.4 Awareness

The switch to knowledge-based problem solving can be influenced by feelings of uncertainty or worry (Reason, 1990) factors that are described by other researchers in terms of ***awareness***. Allwood examined this in the context of evaluations of completed work that are undertaken based on the ***suspicion*** that something is wrong, while the Italian researchers describe it as ***mismatch emergence,*** which is coupled with the understanding that one is responsible for the erroneous action (Rizzo, Ferrante, & Bagnara, 1995).

Stable Frames: Expectations and assumptions about intended actions are not changed during performance. The frame of reference remains the same.	I did some computations with a calculator. I manipulated the data by following a formula kept in my mind. The final result did not seem correct to me. I remade the computation two more times and both results were the same, but different from the first. These latter results sounded right to me. Actually, I did not discover what the error was but only that I had made an error. (E11)
Shifting Frames: Knowledge is updated after executing actions. The frame of reference changes after completing an action, and original expectations are adjusted in terms of outcomes.	I had to make many Xeroxes in the shortest time. I prepared the sequence of articles. I put the sheets over the machine and collected the copies in order to rearrange them in "papers". I was Xeroxing a long paper when I noted that I had to reorder all the copies, because I was feeding from the first page on. Then, I realised that starting from the end of the paper would have spared time and work. (E12)
Distant Frames: The active knowledge relates to a context distant either conceptually or in time or both from the erroneous action.	I decided to clean the luggage rack of my car. To ease the access to the hollow I removed the rear panel bus. Since in the panel there were the speakers of my stereo, I disconnected cables. The day after I turned on my stereo car: the left speakers did not work. I thought it was a fault in the system. One week later, I was in the car talking with a friend the possible causes of the left speaker's breakdown. I recalled that some days before I had my car in a garage to fix minor faults. The guys in the station could have forgotten to re-connect some electric cables. Then, suddenly, I remembered that I had put my hand on electric cables too… (E13)

Lack of meaning. There is no goal governing the ongoing activity.	I intended to pick up the keys of the car. They are usually in a box at the entrance. Instead, I entered another room and searched in a drawer where I did not find any keys (but there were the documents about which I was talking before, but I did not pay attention to them). Then I found myself wondering what I was looking for and why I was there. I had to go back to my office before to recall that I was leaving and so I needed the keys of my car. (E14)

Table 3.2 Frames of reference during action. Adapted from (Rizzo et al., 1995).

The Italian researchers also describe awareness during error handling in the context of active expectations, the ***frame of reference*** held by a person performing an action. Though it shares features with Reason's description of local problem solving or Norman's description of the activation of schema based on the "goodness" of matches (1981), the notion of frame of reference was developed to counter the prevalent view in error detection studies that knowledge is static, or that all knowledge necessary to complete a task is "always available and ready to be used" (Rizzo et al., 1995, p. 8).

Instead, they argue that knowledge, and by extension the active frame of reference, is updated during interactions with an environment. Internally, this is done through the selection of alternative knowledge that is "more appropriate" and externally through new knowledge generated by assessments made of the changing state of the environment. Drawing on related research in psychology, the authors identified four frames of reference, for which they provided examples from their data, given in Table 3.2.

3.4 Summary

The causes of accidents are present in a system long before catastrophe occurs, or a clear sequence of events leading to the accident can be established (Reason, 1990). The notion of latent problems is familiar within software engineering, perhaps made prominent most famously by Brooks. In his description of of software development projects, disastrous schedule slippage is gradual, due to "termites, not tornadoes" (Brooks, 1995, p. 154). Disaster should be preventable. Often there are warnings of an impending disaster, and

some latent conditions should be able to be spotted and fixed (Reason, 1990). However, within software development projects, "day-to-day slippage is harder to recognise, harder to prevent, harder to make up." (Brooks, 1995, p. 154).

Disastrous events are likely never to occur again and so it is necessary to look beyond failed outcomes and to examine the particular details of situations (Rasmussen, 1990). Researchers need to understand how informants select and view events (Crandall et al., 2006). The key is to understand error from an informant's perspective, to reconstruct the view they have when encountering things that go wrong by "standing" in the same situation. The emphasis placed on "local rationality" retrains analytical focus from judgement to dynamic factors that influence performance, including knowledge, mind set and goals (Woods & Cook, 1999).

Software development has been shown to include kinds of work associated both with active and latent error categories (Curtis, Krasner, & Iscoe, 1988; Pennington & Grabowski, 1990). Work is required that cuts across different kinds of tasks, and must be performed in response to higher level organisational concerns. If error is studied in the context of the space of possibilities within which developers perform and not in terms of the overt effects their actions have on software performance, the lens is shifted, from ends that might include critical failure or costly redevelopment to the means that make up everyday practice.

This view affords at once a narrower and a broader perspective. It is narrow in that focus is taken from software in operation, from the organisational or methodological environment in which it is produced, and from the artefacts of which it is comprised. Focus is given to the actions performed by individual developers to create software. It is broader in that analysis of individual actions permit a more general examination of error in the

context of work at the desk, but also in other contexts that depend on different kinds of tools, and that produce different outcomes.

4. Method

For as long as there has been software engineering, there has been error. It is a defining marker, transcending nations, regions and organisations. Research and methodology have been devoted for decades to eradicating, to minimising, to preventing it. Tools are built to manage error; records are kept to track it. Circles of people form in teams, in departments, in companies and governments to look for error, to talk about it, to plan for it, to fume and worry. Individual developers spend hours and hours hunting errors down and getting rid of them.

The problem of error in software development lies with the people who make it. Developers tinker, they are incompetent, un- or improperly skilled, they do not adhere to process. If only developers would build correct software, error would go away. If only they could design and build the right defences, error could be tolerated and the problem of error would go away. If only designs were better, requirements more clearly defined, if development tools were better and easier to use…If methodology and practice were more social, if only developers were better trained, they could get a handle on it, and the cost of error would go down.

This reduction of decades of software engineering research and commentary into two sensational paragraphs was written to provoke a sense of unease (Hammersley & Atkinson, 2007), of strangeness about the relationship between developers and error. Everyone "knows" that the problem of error in software is people, however little is understood about what developers on the job make of it. An ethnographic stance has been taken to explore their perspective. The following pages of this chapter explain what this means.

4.1 Research Focus

Research began with a survey of software engineering research and trade publications that treated concepts related to error and failure. The outcomes of this exercise, selections of

which are reported in Chapter 2 had three immediate, significant methodological implications for the research design.

First, the methodological approach that was initially, naively, proposed was to be ethnographic in that it would examine the topic of error after immersive fieldwork within a single company. The survey of empirical research quickly revealed that with some exceptions (Prior, 2011), long-term, open, immersive access to developers is rare (Easterbrook, Singer, Storey, & Damian, 2008).

Second, at the outset, the starting point for analysis was assumed to be source code and records related to bugs because this is what developers produce and this is where errors are reified (Avižienis, Laprie, & Randell, 2004). It seemed reasonable to assume software could and should be read for evidence of the "tinkering" that goes on during development, to get at an understanding not only of how, but of why it works as it does (Mahoney, 2008).

This is an approach that has been used to good effect, demonstrating among other things how developers navigate within source code (Lawrance et al., 2013), how they engage with APIs in companies (de Souza, Redmiles, Cheng, Millen, & Patterson, 2004) or how programmers use comments to organise and communicate aspects of ongoing work (Storey, Ryall, Bull, Myers, & Singer, 2008).

However other studies demonstrated clear failings in the software records that are kept about error (Aranda & Venolia, 2009), that matched calls for future research consistently made in root-cause analyses. Root-cause analyses studies largely draw upon bug and maintenance reports. Errors that appear in early stages of a project, with less experienced programmers, or after a "hectic period of changes" (Endres, 1975, p.328) are not well represented. Studies have recommended that data about errors should be collected from the entire development cycle, not just at points of testing and integration (Perry, 2010), and

should not be collected too long an interval of time after events have passed (Perry & Stieg, 1993).

Third, the root-cause studies consistently suggested that future research should examine "human erring", including factors such as problems of understanding (Endres, 1975, p.331), inexperience (Perry & Evangelist, 1987), lack of information (Perry & Stieg, 1993), and skill mismatch (Leszak, Perry, & Stoll, 2002). This call matches recent interest to counter technically "saturated" curricula in software engineering with examinations of of engineering process as a "human activity". (Capretz, 2014).

This review led to three decisions:

- Fieldwork would have to be undertaken ***opportunistically, in multiple environments***.
- Examination should ***establish a fuller chronology for error*** by examining activities throughout the development cycle.
- In order to respond to the call to examine "human erring", ***individual experience*** should be the focus of analysis.

4.1.1 The Ethical Impetus

Other concerns shaped the research design. Data is never "pure" (Hammersley & Atkinson, 2007), but contamination seemed to be of particular concern in the context of error. Developers might change their behaviour if they were watched (Hammersley & Atkinson, 2007). Spoken to after a period of observation, they might swagger or boast in their responses (Hammersley, 2003) and not be credible. Organisations might not grant access or treat developers who agreed to partake poorly after the fact.

To address these worries, the responsibility of beneficence as described by Vinson and Singer (Vinson & Singer, 2008) and vulnerable stirrings (Behar, 1997) provided the best guidance. Researchers need to consider potential harm toward companies, ensuring, for

example, that important trade secrets are not disclosed. This can generally be managed in the way findings are reported. Beneficence toward informants is not always so straightforward.

Social research can change the environments in which it is conducted and it can have effects on the people and cultures that are examined (Hammersley & Atkinson, 2007). Outcomes can be put to uses after research is completed that researchers cannot control (Spradley, 1979). These factors were of particular concern during the early stages of this research. Depictions of "incompetent" developers in the software engineering research invoked the spectre of Reason, finger extended:

> *"For those who pick over the bones of other people's disasters, it often seems incredible that these warnings and human failures, seemingly so obvious in retrospect, should have gone unnoticed at the time. Being blessed with both uninvolvement and hindsight, it is a great temptation for retrospective observers to slip into a censorious frame of mind and to wonder at how these people could have been* ***so blind, stupid, arrogant, ignorant or reckless****." [Emphasis added] (Reason, 1990, p. 214).*

Thus chastened, the aim was formed to find a way to perform an analysis of error that would keep ethical concern for developers at the fore. Credible sources of data were sought that would allow observations of practice and interviewing but in which in the research presence would not be considered a threat. Methods were sought to encourage developers to be open and straightforward in their behaviour, and also to ensure that they would not be censured by colleagues for doing so.

The next sections describe how these aims were met, first by establishing epistemological commitments to using ethnographic principles. Next a description is given of gathering material from multiple sites. The process of organising data into sets for analysis is described, and an overview is given of the methods used in individual studies to build up a view on error.

4.2 An Ethnographic Stance

Software development, is, by its nature, socio-technical (Winograd & Flores, 1987), and well suited to an analysis that takes a relational look at human error. Software engineering researchers have long argued for looking more closely at the human aspects of erring related to "knowing", safety science for a naturalistic examination of error using data collected from fieldwork (Le Coze, 2015). To address these calls, this research has taken an ethnographic stance. In this section, a brief overview of what ethnography is and how it is done is given. The section also includes a brief description of uses of ethnography that have been developed in computing research.

Ethnographers study people's actions and accounts of actions in everyday contexts (Hammersley & Atkinson, 2007). The aims of research are often exploratory, beginning with only the sense of a "foreshadowed problem" (Hammersley & Atkinson, 2007, of Malinowski) that will focus over time. Doing ethnography is described as examining "shared order" (Van Maanen, 2011, p. 18), by getting at the means and methods by which people conduct themselves in a social situation. Examination may focus on routine, on ritual, on problems, on the rules or decisions that guide and punctuate action (Hammersley & Atkinson, 2007).

Order is sometimes described as a "mundane feature of everyday life" that serves as the basis for social interaction (Crabtree, Tolmie, & Rouncefield, 2012, p. 162), but this research more closely aligns with the notion that features of order can be identified by examining moments of ***change***:

> *"In picking their way through the minutiae of routine action, prominence is (endlessly) given to the innovative, the ad hoc, and the unpredictable…"* (Anderson, 1997, p. 20)

Ethnography can be identified in terms of how it is done, but also in terms of the stance from which it is performed. The ethnographic mentality entails interpreting meaning

through interaction with and observation of social settings. It is both a commitment to using field work to gather data and the creation of a *post hoc* account of what was seen (Anderson, 1997). Analysis is ***relational***, not causal, by which it is meant that the researcher comes to conclusions without “jumping”, examines appearances in detail, while not accepting those appearances at face value. Likewise, people’s views are considered without making assumptions that they are true or false. The ethnographic stance “pays heed” to things that people may not notice themselves, and may not agree with (Hammersley & Atkinson, 2007).

The romantic view of ethnography holds that fieldwork entails the collection of data through long-term immersion within an environment, such as in a year spent in a village of people who live on a distant, sun-drenched island. But fieldwork has come to be viewed in different terms. Often it is or needs to be conducted in a site or sites (Van Maanen, 2011) that are closer to home, to examine cultures that are familiar (Spradley, 1980), in a process that is contingent (Crabtree et al., 2012) and opportunistic (Hammersley & Atkinson, 2007).

Research is generally not conducted to a fixed and detailed design. The advice given to researchers, while not quite "seat-of-the-pants" (Van Maanen, 2011, p. 74), is ***expansive***:

> "[Y]ou should not worry about where to start: you should start anywhere you can." (Crabtree et al., 2012, p. 95)

It is generally accepted, then, even within social or cultural anthropology (Horst, 2009) that data may be gathered from fieldwork in "any form" (Anderson, 1997), and may be drawn from a range of sources and sites (Van Maanen, 2011). Two conventional views persist: most data will be gathered by participant observation and “relatively” informal conversation, and the time spent in the field with informants must attain "depth" (Hammersley & Atkinson, 2007).

Analysis in ethnographic research begins at the point of collection as researchers formulate ideas about what else might be needed to answer a research question. Interpretations are made by "creating a path" through the data, while reflecting upon different possible meanings (Hammersley & Atkinson, 2007). The interpretative process is ***reflexive***, shaped by the researcher's own experiences and orientations. It is also pragmatic, in that aspects of the field of study or research process may invite or demand the use of different or multiple analytic techniques, uses of theory, and kinds of data (Hammersley and Atkinson, 2007).

Ethnography is both the fieldwork and also the account that is made of the field work (Anderson, 1997). The meanings formed in analysis must be forged through writing: field notes, transcriptions, descriptions and ultimately the reports. The account may be realistic, confessional, or impressionistic (Van Maanen, 2011), drawing together descriptions of people, of settings and processes, elucidating concepts, themes, and typologies that exemplify the social world that has been examined. The account must be authentic, it must inform and illuminate, but must also be authoritative. It must ***convince*** the reader of the legitimacy of what was seen:

> *"[T]o be taken seriously, you have to have been there, seen them, and if not done it and brought back the T-shirt, at least captured and recorded their lives…"*(Anderson, 1997, p. 6-7)

The next section includes a brief description of three uses of ethnography prevalent in computing research. Following this, one use of ethnography to which this research most closely aligns is described in more detail.

4.2.1 Ethnography of, for and within

In two articles written in 1997, ethnographic methods were found to support research and practice in computing in three ways. Beynon-Davies coined the terms Ethnography ***within,***

for and ***of*** in a survey examining information systems research (Beynon-Davies, 1997). Anderson similarly described three uses of "technography" to support system design and human computer interaction research (1997) as ***Integration***, ***Complementarity*** and ***Independence***. Though Beynon-Davies' survey considered a broader number of sub disciplines of computing, the categories of ***within, for*** and ***of*** are sufficiently descriptive to represent the taxonomy given in both.

Ethnography within development employs an ethnographic approach to systems development tasks like design, requirements elicitation (Martin & Sommerville, 2004) or training. The ethnographer in this case is a member of the team (Anderson, 1997). He may perform duties concurrently with development tasks, employing "quick and dirty" techniques or using ethnography to assess designs or specifications with users (Beynon-Davies, 1997, of Hughes).

Ethnography for development produces accounts of how work is done within domains as a way to inform and influence how systems are developed. Anderson makes the point that the specific aim of these studies is to raise awareness or “sensibilities" about the environment in which the technology under development will be used (Anderson, 1997).

Ethnography of research aims to remain independent of design, studying developers and development workplaces. It provides detailed information about the "problems and practicalities" (Beynon-Davies, 1997, p. 537) that arise in creating software. It might illuminate, for example, how developers adapt methodology to the demands of practice, the values given to different kinds of development tasks and the broader cooperative aspects of development.

4.2.1.1 Knowledge is Cultural

That knowledge is cultural and socially produced is a central theoretical assumption made by the uses of ethnography described in both surveys. Studies may elucidate how tacit

knowledge is employed in work, or describe details of practical or “articulated” work in particular settings (Suchman, 1987). These theories underpin interpretations of how workers perform their duties, how they use, or fail to use and adapt technology to fit the requirements of their tasks.

Humans routinely perform skilled activity, but cannot always articulate how they do it. The skills they use are tacit, implicit (Smith, 2003). Beynon-Davies suggests that the concept has been interpreted within ethnographies ***for*** as the knowledge that is required for individual workers to adapt their practices to those of others in a work environment. He links the interpretation to the concepts of ***explicit*** and ***activity*** perspectives on work, developed by Sachs (1995). The explicit view relates to organisational tasks, as represented by defined tasks and procedures. The activity view is socially mediated by workers, through relationships and communication and coordination practices that often involve interaction with artefacts and tools, such as paper-based forms, drawing tools, and spreadsheets.

Beynon-Davies finds three thematic strands of relevance to information systems developers. First, there is the notion the existence and character of tacit knowledge should be considered in participatory design exercises. Second, tacit work practices may have an impact the integration of new technology into "everyday" work settings. The last suggests more generally that tacit work practices underlying cooperative work are *situated* (Beynon-Davies, 1997).

Anderson's survey explores in detail situated work, analysing the development of ***ethnographies for*** design in terms of Lucy Suchman’s study of photocopier failure (Anderson, 1997; Suchman, 1987). He identifies two innovative aspects of her research methodology: First, she re-orientated conversation analysis from its standard use for examining how two people interact, to examining how humans interact with machines.

Second, she used the notion of "communities of practice", or an interpretation of learning drawn from research by Lave as being cultural rather than cognitive (Lave, 1988).

By combining structured, ethno-methodological analytic technique with the notion of socially-based learning, her study was pre-disposed to see the structure and order in these working lives as "situated", "occasioned" and "co-produced" (Anderson, 1997). Though ethnographic studies of technology preceded her work, Anderson argues it was the impact of her methodological stance that galvanised researchers to apply ethnographic methods in the service of design.

4.2.1.2 Technology in Use

The theme of technology as it is used is a second core assumption within ethnographies *for* and *within* technology. Many of the studies examine work practices that depend on computing technology to perform other, "real world" tasks.

This perspective has been widely explored in Computer Supported Cooperative Work (CSCW) research, with studies that examine how employees are affected by new technologies in the workplace (Orlikowski, 1992; Orlikowski & Gash, 1994), the ways in which communities of users engage with collaborative software (Kling & Courtright, 2003), how electronic media support scientific communication Kling, McKim & King, 2003), and how employees use technology to engage with one another (Markus, 1994). The common theme in this research is to study adoption of technologies at the organisational level that have already been developed. While the social environment receives a detailed analysis in this research, the artefacts themselves are often overlooked (Orlikowski & Iacono, 2001).

This view has been interpreted within software development in studies like Randell's description of problems in the development of software for the NHS in England (Randell, 2007), or Ince's analyses of software for supporting social work (Ince, 2010). Their view, like that in CSCW studies, is that domains of work should be considered as socio-technical

in nature. Domains must be studied in these terms to determine how best to make software in their service.

4.2.2 Ethnographically-Informed Research

The case for writing ethnographies *of* making software has been made most clearly by authors advocating for ***ethnographically-informed*** research (Robinson, Segal, & Sharp, 2007; Sharp, Robinson, & Woodman, 2000). According to this view, following ethnographic principles is necessary because the essential nature of work practice cannot be known *a priori*, and cannot be taken as "official".

In contrast to other approaches in software engineering, following an ethnographically-informed approach allows research to be performed that is exploratory and which considers open ended research questions. The intent is to understand something more about the work practices of software engineering ***itself***. It is argued that software engineering forms a culture that transcends national, regional and organisational cultures. Markers of this culture that have been observed using this approach include community and constituency, a lack of importance given to evidence-based practice and argument, and the importance of the role of the local guru (Sharp et al., 2000).

Ethnographically-informed research of software development is performed by researchers who are members of the software engineering discipline. However, the aim is to understand practice in its own terms and not in terms of prior understandings formed through membership in the discipline. The researcher must be "more observant" and "more critical" of the field to which they belong, of "what we do and how we do it" (Sharp, Robinson, & Woodman, 2000, p. 42). Rigour is achieved through triangulation of different data sources and feedback gathered from informants.

Ethnographically-informed research entails making two adaptations to classical ethnography. Both correspond to problems that were also noted by Beynon-Davies in

regards to ethnography that is applied *within* system design. First, in order to meet the constraints of performing fieldwork in software development environments, ethnographically-informed research abandons the notion that long-term, immersive access is necessary (Robinson et al., 2007). Instead, the studies often rely on informal and opportunistic data collection rather than gaining access to participants in a more formal, structured way.

Second, empirical investigations conducted using ethnographic principles are combined with other analytic methods. This strategy allows researchers to explore broad questions of how work is done while responding to the scientific demands of engineering. Rigour is achieved in analysis through the use of methods such as documentary research, discourse analysis and grounded theory. For example, analysing talk provides a method for examining what language is used for, it permits researchers to: "listen to what is being done with the words." (Sharp et al., 2000, p. 42).

Combining analytic techniques is also a concern of the other two uses of ethnography. For example, Ball and Ormerod describe ***cognitive ethnography*** as an approach *for* design that is specific, purposive and verifiable. The approach entails gathering small-scale data from "representational" time slices, research questions are designed to intervene or otherwise affect work practice and the validation of results with observers and using experimental methods to "methodologically" triangulate. They argue that ethnographies conducted to assist design differ and must differ from features of prototypical ethnographies because they are “purposive”, they have applied aims for improving teams or design process (2000, p. 408)

4.2.2.1 Weaknesses

Weaknesses associated with ethnographically-informed studies regard concerns about achieving depth and overcoming membership. Criticism is made that adaptations of ethnography often make use of fieldwork "blitzkrieg" (Van Maanen, 2011, p. 164), that

disciplines are guilty of "do-it-yourself" ethnography. Van Maanen is particularly critical of member-performed ethnographies because of their approach to gathering data, and on the basis that they have not sufficiently attended to the "invisible work" of ethnography (Forsythe, 1999) [3].

However, the need to combine methods and to adapt principles of classical ethnography is pragmatic, and in fact may be necessary to answer particular research questions (Hammersley & Atkinson, 2007). Field work performed without immersive access often must depend on data captured on video or audio recordings. Mixed methods are used in these cases to address demands of the media formats (Heath, Hindmarsh, & Luff, 2010), or to achieve depth in analyses by performing detailed, fine-grained or "micro" analyses (Knoblauch, 2005). Ethnographically-informed studies of development are necessarily ***focused,*** and produced by members, in order to examine specialised and fragmented activities (Knoblauch, 2005).

The mixed analytic approach also has a history within ethnography that predates use in computing research. As noted, Suchman's method of examining interaction was drawn from earlier developments in ethnomethodology (Anderson, 1997). The use of techniques like card sorting (Ball & Ormerod, 2000) was also advocated by earlier ethnographers who used the technique to develop and test the strength of informant created taxonomies (Spradley, 1979).

The substrate underlying ethnographies *for*, *within* and *of* is common: work is done between people and between people and machines, knowledge is social and culturally produced. What is different in ethnographies *of,* and in particularly in *ethnographically-*

3.The criteria of "sufficient attendance" seem particularly difficult to address. It is not clear how one proves, for example, that one has engaged deeply enough with anthropological writings on ethnography (Van Maanen, 2011) Similar points have been argued in the context of adaptations to ethnography employed by Computer Supported Cooperative Work researchers (Bannon, Schmidt, & Wagner, 2011). Bannon et al. argue that it is difficult to determine studies that employ "scenic" ethnography, but agrees with critics that it is necessary to determine that data have been sufficiently "analytically worked".

informed studies is that ***developers*** are the workers to be examined, and making information computing technology is the work that is being done. The emphasis in ethnographically-informed studies is not to further social science agendas, but to return to classical ethnographic aims, examining the actions and interactions of people.

In the next section a description is given of the field sites and sources that were used to inform studies.

4.3 Field Sites and Sources

A pragmatic decision was taken early in the research process to temper collection with ***gleaning***, to look for data within sources that had been created by other people. Gaining access to software development sites is difficult (Easterbrook, 2008), particularly when access is sought to examine mistakes (Perry, 2010). In this research, the vagaries of access were not overcome, they were worked around. Sources were identified opportunistically. Relatively unstructured, open access was gained to sites for interviewing through contacts within standing professional and academic networks.

Data were gathered in a step-wise fashion (Horst, 2009). Step-wise collection has been described in the context of ethnographies that examine trans-national cultural concerns, for example of migrant populations. In these cases, research must, of necessity, be undertaken by a single researcher who travels to multiple sites. In the case of this research, it simply means that data were sourced, gathered and examined from different sites at different times.

Implicit in the decision to glean was the assumption that these secondary sources (McGinn, 2008) would likely take the form of video recordings of practice that could be indirectly examined for evidence of error. With the help of the supervisory team, contact was made with other researchers who had already been able to gain access to professionals and industrial environments. Participant-created video (Hammersley & Atkinson, 2007)

was sought on the internet that depicted professional developers at a sufficient standard of production to permit rigorous analysis.

4.3.1 Sites

Four field sites were used. One set of data was gathered in the Spring, 2010. The rest of the corpus was collected between January 2012 and April, 2013 (see Table 4.1 for a summary). Two departments performing work for universities were represented (Sites B and D). At site A, the pair of designers observed worked professionally as colleagues for the same company. The structure of teams at this company is unknown, but was considered to be inconsequential to analysis.

Site	Context	Method of Collection	Date of Access (Creation)
The AmberPoint Design Session (Site A)	Design, set task (organisational simulation); laboratory	Video recording, gleaned for secondary analysis	2010 (2009)
Digital Humanities (Site B)	Project work, organisational tasks; university	Interview, observation	2012
Acceptance Test Framework (Site C)	Desk work, personal tasks; industry	Video recording, gleaned for secondary analysis	2012 (2009)
Course Planning (Site D)	Project work, organisational tasks; university	Interview, observation	2013

Table 4.1: Field Sites. The Site column indicates a descriptive name along with a letter that reflects the order of access. Sets of data were later grouped for reporting, as depicted in Figure 4.1. Date of access indicates when the data were accessed to support research in this thesis. The date in parentheses in this column indicates the date when sources were originally created.

4.3.1.1 Access

Though one aim for research was to perform indirect observation, access to environments and materials was overtly sought. Four managers working in two organisations (Site B and D, reported in Chapter 7) gave permission to observe and speak with employees. Each interviewed person was given an information sheet about the project, and signed an informed consent form. These materials were reviewed and approved for use by the Open University Ethics Committee.

Researchers working in the Software Design and Collaboration Laboratory in the Department of Informatics at the University of California in Irvine and their industrial partner, granted permission to observe video that had been collected for separate research projects (Site A, reported in Chapter 5).

Two professional developers granted access to examine video that they had created and released to the internet (Site B, reported in Chapter 6). The videos used in analysis were uploaded to a public video site by two professional developers, the terms of which permit free personal use. The creators of the videos gave permission to use the videos for research in a series of email exchanges. The videos feature audio input from other people collocated in the office at the time of recording. These participants could not be contacted; it is assumed that the creators of the videos obtained permission before recording and uploading the videos to the hosting site.

4.3.2 Corpus

The full corpus includes proprietary and participant-created video, audio recordings, transcriptions of video and interview, and field notes taken during and after site visits. Table 4.2 lists the sources of data collected from each site. Ephemera were collected that include photographs, drawings, diagrams, historic and interview related email messages, screen grabs of social media pages and blog posts, and source code. In the course of

sharing research at workshops and in meetings, a number of anecdotes and observations about personal encounters with error were collected. These were used to identify points of resonance and dissonance, and to hone areas for investigation.

Site	*Sources of Data*	*Description*
The AmberPoint Design Session (Site A)	- 1 video recording, 2.5 hours long - 1 preliminary transcription - 1 enhanced transcription	Set design task, followed by a brief interview in which the designers reflect on the session.
Digital Humanities (Site B)	- 7 audio recordings - 6 transcriptions - Field notes taken after interview. - Photographs of work spaces, design diagrams, email exchanges, snippets of code	Semi-structured interviews collected using an adaptation of the critical decision method.
Acceptance Test Framework (Site C)	- 60 video recordings - 20 transcriptions Blog posts and website information, social media alerts and photographs, and open-source code archive	Depictions of several months of intermittent development on an open-source development project. Analysis performed on one-month subset.
Course Planning (Site D)	- 4 audio recordings - 3 transcriptions - Field notes taken after interview. - Field notes recorded in half-day observation. - Drawings, diagrams, email exchanges.	Semi-structured interviews collected using an adaptation of the critical decision method.

Table 4.2. Sources of Data, by field site. Data were later organised into sets for analysis and reporting (see also Figure 4.1) that relate to their source and media format.

4.3.3 Informants

Fifteen developers informed this research, three females and twelve males. Their shared experiences represent a range of different software development tasks, including high-level design, data modelling, interface design and development, and application development. They also represent diverse working practices including domain-driven development, open-source development, industry sponsored open-source development.

Sampling was opportunistic. No effort was made beforehand to identify people who were considered to be experts or novices, and collection was not restricted to developers who were performing specific tasks. The informants worked for organisations or companies; one pair of developers are professional freelance consultants. Additional details about developers can be read within individual study chapters, in Sections 5.2, 6.2 and 7.2. Each section has the title *Setting the Scene.*

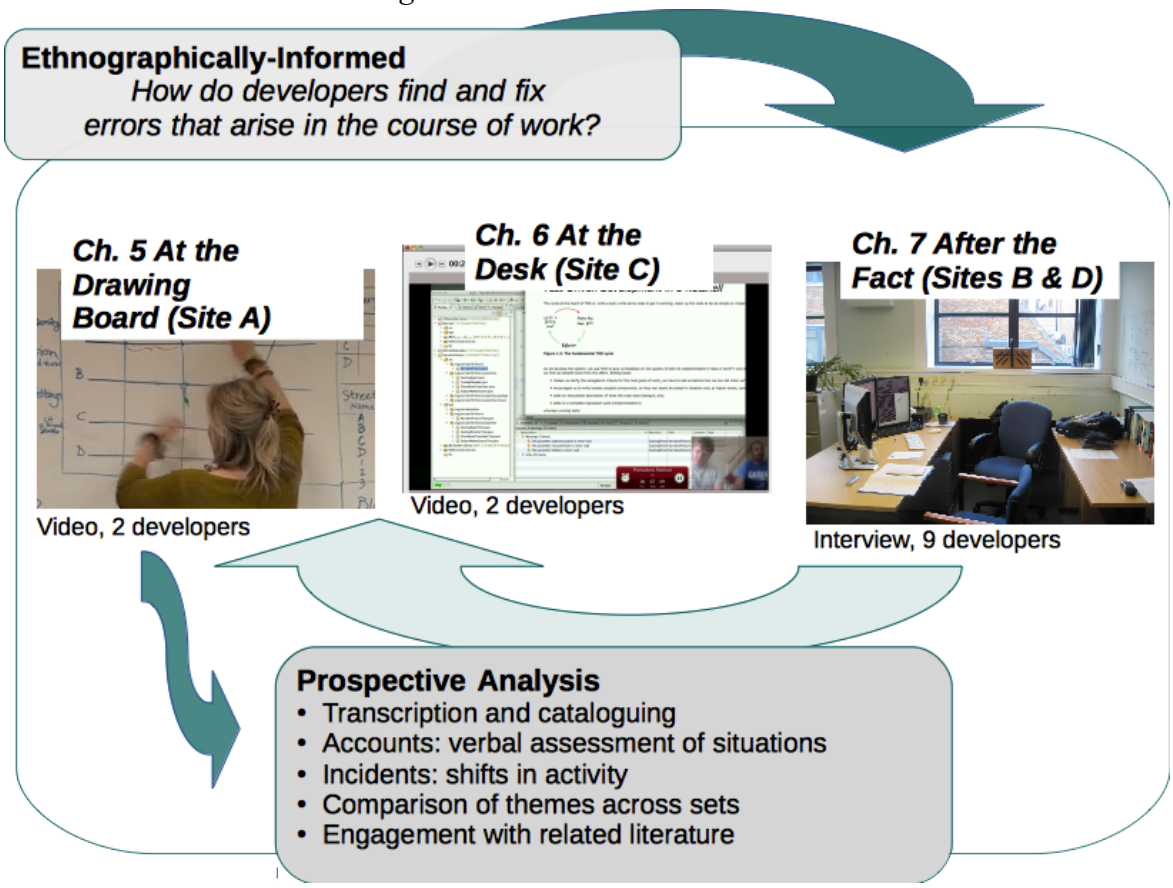


Figure 4.1. Overview of Method and Studies. Sets of data were identified to support three studies. The groupings that were made reflect different periods of practice and also particularities of the data, i.e. whether they were gathered by video recording or interview and by whom. The video of design activity analysed for Chapter 5 was recorded by one researcher, the videos depicting work at the desk in Chapter 6 were created by a pair of developers, and the interviews used in Chapter 7 were collected by the author of the thesis.

4.4 Studies

Research began with the broad question 'How do developers find and fix errors that arise in the course of work'? This question corresponds to principles of error handling that have been examined in psychology and safety science, and in methods and theories associated

with cognitive task analysis, itself a field with links to both disciplines (Crandall et al, 2006).

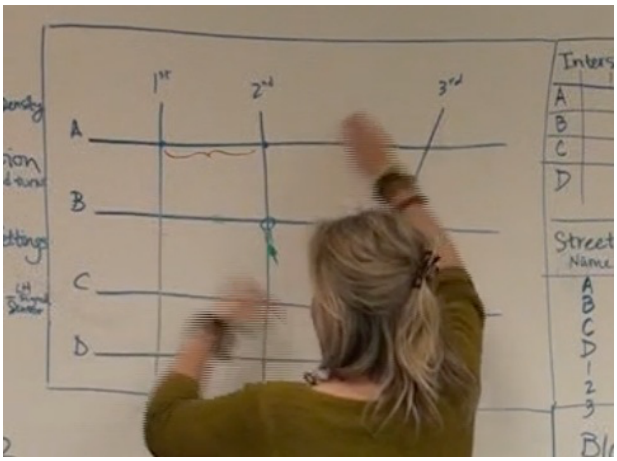
The corpus was formed into sets for analysis using principles of thematic analysis. The groupings that were made are depicted in Figure 4.1, above. This method was selected because it is not overly structured. It is also "theoretically free", belonging to a group of methods that can be used independently of theory and epistemology (Braun & Clarke, 2006). The flexibility of the method made it possible to support opportunistic access to sites, a broader look at tasks and time, and to examine personal experience. It was also possible to interweave examinations of related literatures with identification of themes within individual sets and by comparing instances across sets.

Though the research question preceded knowledge of these related literatures, the interpretation of diverse sources was increasingly focused over time (Hammersley & Atkinson, 2007) by drawing on concepts and models found in them. Iterative assessments were made of software engineering literature, analyses were performed on data, and literatures associated with human error were consulted.

The following sections given an overview of the research process for each study. In each case, details are given about the research interest that let to collection, as are a description of the sources that were accessed, analysis techniques, timeframes. Notes are given about how examination of each set impacted analyses for other studies.

4.4.1 At the Drawing Board (Site A)

The Amberpoint Design Session



- Two developers
- Activity at a white-board
- Set-task, video

Figure 4.2: Overview of At the Drawing Board (Site A).

At the Drawing Board (Chapter 5) explores how designers develop their awareness of problems. It assumes that software development is primarily a design activity (Pennington & Grabowski, 1990) and that difficulties can remain active even after local recovery. The data used in this study was drawn from materials created for for the National Science Foundation funded International workshop "Studying Professional Software Design" (SPSD).

The activities that are depicted in the videos (see also figure 4.2) reflect the kinds of work that are performed before code is written or modified, in settings in which developers do not have immediate access to source code. The study had two aims:

- First, to identify ***indicators of awareness of issues*** in design that might be compared materials that depict problem solving in different development contexts.
- Second, to compare ***problem solving*** as conceived in studies of design with descriptions of problem solving drawn from psychology and safety science.

Analysis began by annotating the transcript of one design session from the SPSD workshop. The AmberPoint session transcript was amended to include information about gesture and whiteboard work, as well as additional linguistic content.

This was followed by segmentation of the transcript to isolate particular incidents for study; each incident was additionally broken down into distinct periods within the session in which the incident was discussed. Incidents were identified selected by isolating topics discussed more than once over the course of the design session. These repeated discussions included elements of the following:

- Re-examination of tentative decisions (Guindon, 1990)
- Attachment to concepts (Cross, 2001).
- Disagreement (e.g. “I don't think so”) or (“I don’t think it needs to be…”)
- Lack of understanding (e.g. "I don't know")
- Lack of confidence, for example signalled by repeated turns away from the whiteboard and the corresponding provision of assent in the form of paralinguistic utterances (e.g. "mm hmm", "yeah").
- Representation difficulties, as indicated through repeated use of problem framing, reference to the design prompt, use of gesture, or extensive re-working of diagrams.

After incidents were selected, individual incidents were transcribed within a columnar catalogue following the conventions given in appendix B.1. The catalogue cross-referenced dialogue with information about:

- Gestures
- Whiteboard work, specifically sketching or amending existing sketches, and;
- Focusing, by making references to the design prompt, or noting longer periods of examination or re-examination of the design prompt.

Within the broader framework of Cross' principles of design cognition, individual exchanges were examined for evidence of the particular kinds of knowledge exploited by designers as identified by Guindon. A catalogue of Guindon’s work has been extracted and can be consulted in appendix B. 3.

4.4.1.1 Timeframe

A preliminary analysis was performed in 2010 to establish parameters for examining problem solving in development activities that don't involve writing code. This analysis also helped establish methods for analysing video and for examining paired interaction. The interpretation of the data was subsequently developed in 2013, and completed in 2015 after comparison with data from the other studies, and through examination of a second set of videos.

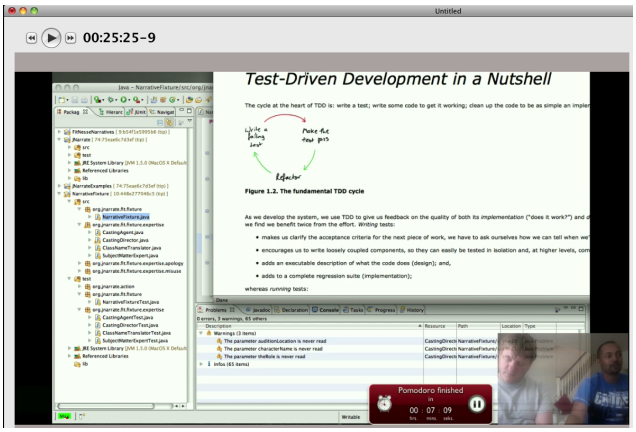
4.4.1.2 Relation to other studies

Analysis of the design session revealed gaps in depiction and scope. The experimental session examined was hypothetical, and analysis of conversation revealed features of discussion about problems that were suspected to be representative more generally of software development, rather than of design activity.

The study formed a baseline for examining problem solving prospectively, providing evidence for visual and verbal signals that were also used during analysis of materials for Chapters 6 and 7. Signals are described in more detail in appendix A.2

4.4.2 At the Desk (Site C)

Acceptance Test Framework



- Two developers
- Development in an IDE, web browser
- Self directed tasks, video

Figure 4.3: Overview of At the Desk (Site C).

At the Desk (Chapter 6) examines how developers interpret the software that they use and write. Focus is placed on moments in which things go wrong in desk work. The data used in this study was drawn from paired interactions between two developers who filmed themselves over the course of a month as they modified an open source tool (for a summary of filming dates, see Figure 4.4)

The aim of the study that was to gather evidence for errors as they occur at the desk, while software is being written. Three goals were set for the study:

- First, ***incidents were to be identified in work prospectively,*** rather than using bug reports or repository snapshots as a starting point for retrospective analysis.
- Second, filmed sessions of paired work that ***spanned a calendar month*** were examined.
- Third, emphasis was given to illuminating ***situational and circumstantial factors in decision making***.

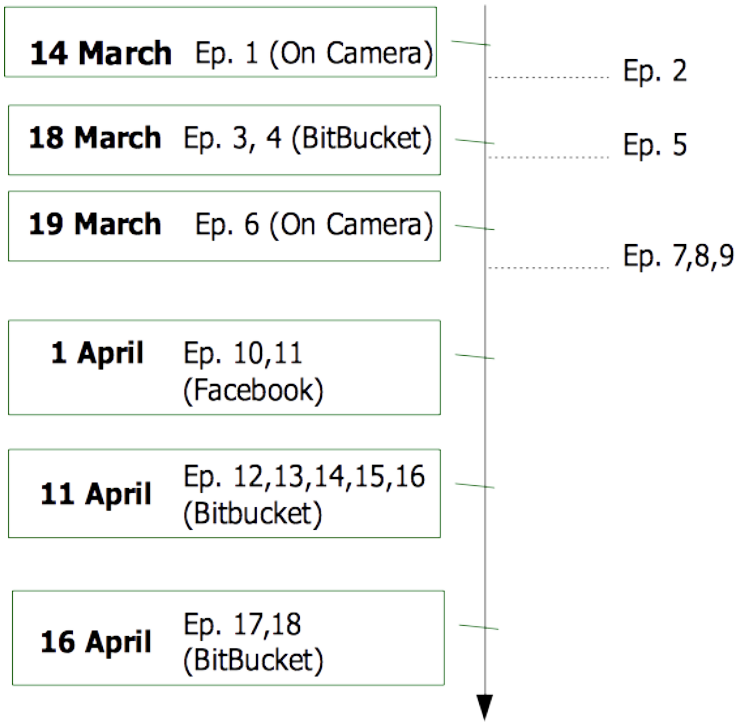


Figure 4.4: Filming dates at the desk in 2009. () indicates the source used to establish the filming date; episodes listed on the right are those for which a date could not be established. A single programming session can span multiple recorded episodes. For example, episodes one and two correlated to two programming sessions, which occurred on different days. By contrast, episodes 3 and 4 comprised a single programming session.

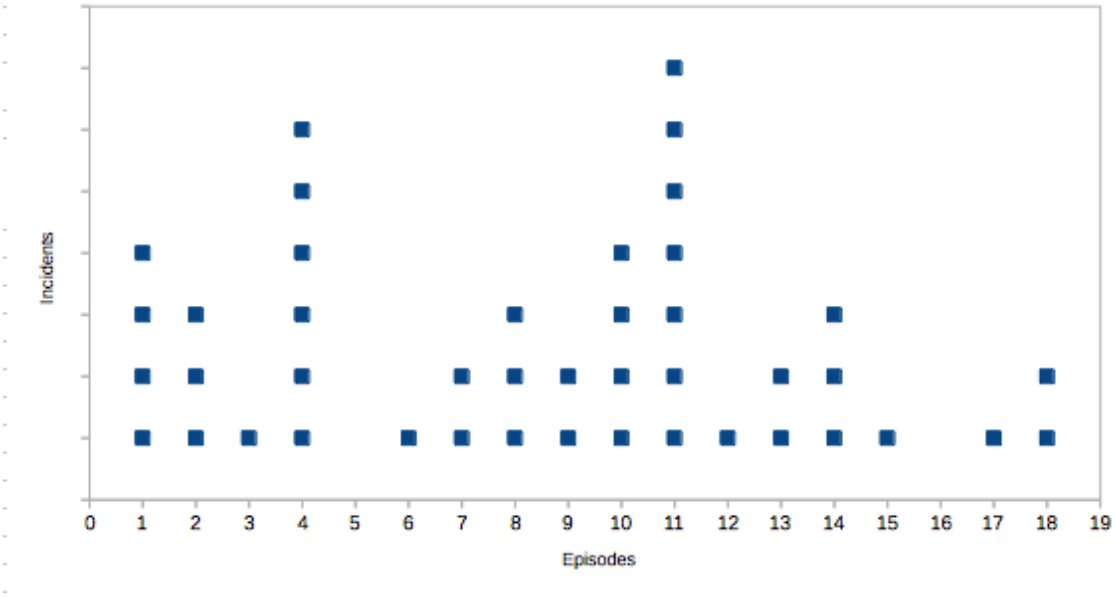


Figure 4.5. Breakdown of incidents at the desk by episode. Forty-three incidents in episodes 1-18 were examined in detail. An additional twenty-five incidents were considered. Eleven were used to develop contextual understanding for practice, while fourteen depicted issues related to conceptual design or to global aims for the project. A catalogue of common attributes for the forty-three primary incidents has been extracted into appendix C.2.

The developers produced sixty videos. Figure 4.4 above, gives an overview of filming dates for the corpus. Transcripts were created of twenty films. Episodes 1-18 were iteratively segmented to isolate incidents for analysis. Episodes 19 and 20 were transcribed and have been used to inform analysis but were not segmented to isolate specific incidents (see also Appendix C for more detailed information about the video corpus, processing and incidents).

Initial thematic analysis of ten transcripts showed that affective factors noted in early analysis of high-level design and in accounts of past work were useful in broadly identifying "curious incidents" (Crabtree, Tolmie, & Rouncefield, 2012) in the videos. The videos contained evidence of surprise, suspicion and of doubt, taken to be indicators of shifts between progressive and evaluative problem solving (Allwood, 1984) They also included examples of local problem solving (Reason, 1990).

However, in many cases, the activities undertaken when errors arose were simple. The developers appeared at times to make mistakes that were related to mechanical or routine skill rather than grappling with conceptual or design problems.

Analysis was subsequently undertaken in three stages, using affective features to draw out material from videos, but also delineating and marking other features to deepen analysis of problem solving. Forty-three incidents were selected for detailed analysis, a figure indicating coverage of the incidents across episodes is given in Figure 4.5.

First: The aim of this analysis was to capture high level details of incidents. A catalogue was created that noted:

- **Chronology:** duration, start- and end points
- **Artefacts:** files that were accessed in the screencast over the course of the incident
- **Roles:** determination of which programmer was working at the keyboard
- **End state:** resolution at incident completion, relation to other incidents,

- **Recovery:** the actions taken to recover were noted (e.g. changing wiki syntax or altering a tool configuration)
- **Detection:** mode (e.g. outcome- or action-based), What was said, source of information (e.g. system response), corroborative verbal and visual indications

Second: In this analysis, the circumstances surrounding error detection for a subset of the incidents were examined. To do this, the dialogue and screen activity associated with detection were catalogued in more detail. The sequence of a subset of incidents was also diagrammed by hand to explore the handling process. The catalogue included the following information:

- **Insight:** the source of information that provided information (e.g. something seen on the screen, prior experience, what was said)
- **Handling**: notes of strategies, tactics, evidence of guessing, trying things, and manipulations made to the environment
- **Recovery mechanism**: how the recovery was achieved (e.g. changing syntax, flushing a cache, altering method calls or class signatures)
- **Number of attempts**: a rough assessment of the number of attempts that were made before recovery was achieved
- **Rate of understanding:** indications that one developer figured out the problem before the other

Third: Exchanges within twenty-five incidents were segmented and coded line-by-line to develop understanding about local cycles of problem solving. The aim of this analysis was not to develop a fixed model, but to gain a better sense for how stages of detection, identification and recovery are interleaved by tactical approaches, manipulations of the environment, and emotions that modulate the process. A portion of one incident that was coded this way can be seen in Appendix C, Section C.3.3.

4.4.2.1 Timeframe

Near-verbatim transcripts were created of twenty videos in 2012 and early 2013; these were iteratively segmented and catalogued to isolate incidents for analysis. Sixty-eight incidents were analysed, initially in 2013, with subsequent interpretation in 2015. Preliminary findings were presented at the PPIG 2015 work-in-progress meeting (Lopez, Petre, & Nuseibeh, 2015).

4.4.2.2 Relation to other studies

Analysis of high-level design activity and accounts of recent work revealed gaps in depiction and scope, which led to the analyses of work at the desk. The second analysis revealed that error handling in software development is often cyclical, involving more than one round of problem solving. These observations were used to undertake a more detailed review of the psychology literature related to human performance and error detection, the outcomes of which were reported in Chapters 1, 2 and 3.

4.4.3 After the Fact (Sites B and D)

Digital Humanities and Course Planning

- Six developers
- Solicited reflection about recent work
- Organisational tasks, audio

Figure 4.6: Overview of After the Fact (Sites B and D).

After the Fact (Chapter 7) examines accounts given by programmers about problems encountered in recent work (see the overview in Figure 4.6). Software takes time to write

and experiences with errors are personal. The significance of errors will reflect passing time and social and organisational influences. The data in this study was drawn from semi-structured interviews conducted at two field sites. The aims for this study were two-fold:

- First, to gather evidence about error from the ***full development cycle*** in professional contexts
- Second, to give developers an opportunity to identify and describe incidents of error handling ***in their own terms***

Eleven individuals were interviewed in computing departments at two universities in the United Kingdom. Each informant was asked to recount an incident from recent work in which they played a discrete role. Interviews were gathered following an adapted protocol of the critical decision method, a technique used in cognitive task analysis to study how decisions are made in real world settings (Crandall, Klein, & Hoffman, 2006). An overview of the method is given in Section 4.5.4.1. A detailed summary and commentary on the protocol that was applied can be read in Appendix D.2

Informants were sought opportunistically at site B and D; acknowledged experts were not identified beforehand. Meetings were arranged in person or by email, and each person was sent an information sheet before the appointment (see Appendix D.4). The information sheet was reviewed with the informant before the conversation, and each person signed an informed consent form. Interviews were audio-recorded, and notes were taken, in sessions that lasted from between forty-five and seventy-five minutes. Interviews concluded with questions about training and experience.

Nine interviews were selected for analysis and transcribed. Six interviews were analysed for evidence of error handling. Each was selected because it included sufficient detail about ***what the informant did*** to detect, identify and recover from their problem. It

also provided information about how the informant developed ***awareness of the problem through these stages***. Three additional interviews taken at Site B were used to inform contextual understanding; additional explanation is given for exclusions in Chapter 7, Section 7.2.4.

Transcripts were first coded into segments. Segments were identified by questions and responses that moved discussion in a distinct direction; this determination was made by assessing how an area of the transcript broadly corresponded to areas of questioning. The critical decision method entails examining a single incident in four semi-structured "sweeps" (for a fuller description of the protocol, see Appendix D.2) Each sweep is used to elicit details about decision making from different perspectives:

> ***Identification and Accounts*** -In this sweep, the informant and the researcher identify a critical incident, and the participant gives a brief account of what happened. The participant provides the structure of the interview, through the content of the story and the details they provide about sequence, beginning and end points. The person must recount a story in which they were a "doer" or decision maker, and the interviewer must help establish what kind of story is representative within a domain and relevant to the research problem.
>
> ***Juncture in Time or Decision Point*** -a timeline is established to note critical decision points. A critical point is one in which the informant experiences a major shift in thinking or understanding about a situation, or takes decisive action. They are critical in the sense that they are "turning points" at which different decisions or actions may have been taken (Crandall et al, p. 76).
>
> ***Deepening -***The process of establishing a timeline interleaves with a more detailed recounting of the incident itself. In the process, deepening probes are used to elicit information about cues and patterns the participant perceived, the rules-of-thumb they devised, the kinds of decisions they had to make, and details about particular cases.

Hypothetical Alternatives - each participant is asked to consider hypothetical alternatives to decisions that were taken, or to consider how someone else might have handled the incident.

Each segment was coded to reflect themes in the data. Individual segments often included more than one question and response and almost certainly included information relating to more than one category. Multiple categories were often assigned to reflect evidence of more than one area of deepening, such as a response that described information that was sought, and how that information related to goals or priorities. A fuller description of the coding process can be read in appendix D.3.

4.4.3.1 Timeframe

Interviews at Site B took place in the Winter of 2012. Interviews at Site D were conducted in Spring 2013. The lapse in time was largely due to opportunity. It simply took longer to locate and negotiate access to a second organisation. Transcription, segmentation and initial coding for interviews at Site B were performed in 2012. Interviews from Site D were transcribed and segmented in the Spring of 2015. The codebook was developed and applied through subsequent thematic analysis undertaken in Spring and Summer 2015.

Preliminary findings of thematic analysis for interviews collected at one site were presented at the CHASE 2012 Workshop (Lopez, Petre & Nuseibeh, 2012-a) and at the PPIG 2012 (Lopez, Petre & Nuseibeh, 2012-b) yearly meeting.

4.4.3.2 Relation to other studies

As in qualitative examinations of other fields (Allwood, 1984; Orr, 1986), the first analysis of accounts of error suggested that problem solving during error occurrence may be lengthy. The amount of time required for identification and recovery may have effects that are felt more or less immediately but which take longer to resolve.

Initial analysis of interviews revealed that the degree of precision and depth of the information in the accounts was different, particularly in relation to what was reported about what had been done at the computer, and what had been done in the past. This was interpreted to mean that accounts must be paired with other forms of data, and in particular data that could show development of code in real-time, and over time. It led to the identification of the data used in Chapter 6, At the desk.

The lapse in time between site visits was an effect of gaining access, but also served broader aims. Interviews taken at the second site allowed the critical decision method to be applied a second time and to differentiate findings related to particular domains or environments from those that might more broadly characterise general aspects of software development.

4.5 A Prospective Analysis

The methods used to guide analysis were selected because they are ***prospective***, allowing actions to be followed forward in time**.** The approach taken toward interpretation has been "semantic" (Braun and Clarke, 2006, p.81). Analysis has not looked "beyond" what is said by the developers themselves. Rather than using accounts to understand values, assumptions or social relationships (Sharp, 2000), they have been considered in the narrative sense (Orr, 1986), as sources from which to identify points in time, including chronology and sequence, and to define components of problem solving in relation to interactions with machines and between people.

In the following sections, more detail is given about general principles and methods that were used to support analysis, beginning with a review of analytics used in relevant psychology and safety science studies, followed by approaches taken to qualitative analysis in software engineering research.

4.5.1 Related Approaches

Data for the sense of error described in these pages relies on evidence gathered from observation or that is perceived and reported, at or soon after occurrences. Ideally, the data should be gathered under naturalistic conditions. Studies of human error in psychology have used think-aloud protocols (Allwood, 1984), but have primarily relied on reports collected from diaries (Norman, 1981(Reason, 1984; Sellen, 1994). Studies in safety science, which more commonly examine error in the workplace, have used think-aloud (Rasmussen & Jensen, 1974), time-slice observation and interview (Zapf, Brodbeck, Frese, Peters, & Prümper, 1992), and simulations performed by computer users (Rizzo, Bagnara, & Visciola, 1987) steel plant (Rizzo, Ferrante, & Bagnara, 1995)[4] and nuclear plant operators (Woods et al., 1994).

The limitations of these methods are evident. Self-reports may be inaccurate, incomplete or biased (Reason, 1984, p. 520) while protocols captured using think-aloud, often in experimental settings, may be forced or artificial (Miyake, 1986). Interviews are difficult to conduct, and depend upon willing, articulate informants. Observation is costly and hard to arrange in professional settings, and it may be difficult to focus in the moment on the significant aspects of the work that is being performed (Crandall et al., 2006).

It is generally recommended that methods be used in combination as resources allow (Crandall et al., 2006) and as research questions demand (Hammersley & Atkinson, 2007). Interviews, for example, commonly provide a practical complement to observational data (Hammersley, 2003), as the data collected in each can be used to "illuminate" the other (Hammersley & Atkinson, 2007, p. 102). In safety science, interviews taken after a period of observation have been used to enhance or refute understandings about human error formed through observation alone (Hollnagel & Amalberti, 2001). This is good practice,

4. The study of steel workers that is commonly cited for these researchers could not be accessed. Information about it has been drawn from this later analysis, from (Rizzo, Parlangeli, Marchigiani, & Bagnara, 1996) and by a summary made in Reason, 1990.

because all of the information relevant to research is often not available "first hand" (Hammersley & Atkinson, 2007, p. 98) in observation.

4.5.1.1 Software Engineering

Interviews and observation are often used in software engineering research to fill in gaps of understanding of practice gathered by examining other evidence, such as source code, records stored in tools, in bug reports or maintenance requests. It is increasingly accepted that the "human aspects" of software development cannot be understood solely through analysis of outcomes (LaToza & Myers, 2011). At the same time, the challenges of working with unstructured, qualitative data have also been observed (Kitchenham et al., 2002). Aranda and Venolia describe mixing methods to balance the need to collect large amounts of rich, contextual data with that of conducting focused analyses (Aranda and Venolia, 2009).

Studies that examine practice commonly use data drawn from tasks undertaken at the desk. For example, Bowdidge & Griswold (1997), Ko and Myers (2005), Lawrance et al. (2013) and (Park et al., 2013) examined video recordings of developers working on set tasks. They based their analysis on verbal utterances and other information drawn from the recordings, and either traced actions backward in a process described as "basically debugging" (Ko & Myers, 2005, p. 62) or forward in time to identify behavioural cues such as verbalisations, reactions and strategies (Park et al., 2013) or evidence of "foraging" for information (Lawrance et al., 2013). Errors that people make while writing HTML code have been examined by examining In these and other similar studies (Huang, Liu, & Huang, 2012), interpretations of actions were made in relation to items from classifications adopted beforehand, and used a definition of error as material, "fragments of code" that are left behind (Ko & Myer, 2005). Likewise, the emphasis in both was to model in general terms how developers reason during tasks associated with bugs with aims to suggest

improvements to support use of tools or to make suggestions for helping programmers write code that is better suited to bug fixing tasks.Though it has different aims, the research reported here has used similar sources and methods to examine error as it is encountered. However, rather than working backward from outcomes to examine past actions (Ko & Myers, 2005), analysis has more similarities with the prospective analytic technique described by Bowdidge and Griswold.

4.5.2 Transcription and Cataloguing

Transcription is part of the interpretative process, but it was also central in this project as a way to draw materials together for analysis. The opportunistic approach taken toward collection resulted in a large corpus of unstructured data in various media formats. The videos used in analysis were filmed by other people in diverse settings. Limitations in the data brought by having had only "mediate" access to the events they depicted (Scott, 1990) were overcome by creating a set of familiar texts from which to perform analyses.

Though direct, contemporaneous access was not possible, it was possible to treat the videos as a sort of "borderline" document between a record and aesthetic material, and to witness many of the audible, visible and tactile aspects of the action that were depicted (Scott, 1990). The videos also might be interpreted as including inscriptions of other texts: of the software that is being written, and diagrams and text that is written on whiteboards. These other texts were considered to be peripheral, in that emphasis in transcribing all sources has been to identify speech-based text.

Audio recorded interviews, design video and programming videos were transcribed using the same basic transcription conventions, defined to capture details of speech and interaction. Conventions were adjusted to meet requirements of different media. A fuller description of the methods used to transcribe and process materials can be read in appendix A, while details particular to individual studies are located in appendices B, C, and D.

4.5.2.1 Interaction Analysis and Focused Ethnography

To learn how to identify and manage sequences of audio and video recorded activity, principles of interaction design (Jordan & Henderson, 1995), videography (Knoblauch & Tuma, 2011; Knoblauch & Schnettler, 2012) and more general descriptions of qualitative analysis using video (Heath, Hindmarsh & Luff, 2010) were studied. Capturing data for observation on video recordings produces data that is by some accounts more objective (Knoblauch, 2005). Recordings can be gathered less intrusively (Jordan & Henderson, 1995), allowing researchers the opportunity to "look again", and also to perform detailed "micro" analyses.

In a recent survey, Knoblauch describes ***focused ethnography*** (Knoblauch, 2005) or ***videography*** (Knoblauch & Tuma, 2011; Knoblauch & Schnettler, 2012) as a "distinct" form of ethnography adopted in applied research. An earlier description of performing focused, micro analyses of video recorded material was described Jordan and Henderson as ***interaction analysis*** (1995).

Interaction analysis holds that cognition is socially oriented and distributed, "situated in the interactions among members of a particular community engaged with the material world" (p. 41). In practice, it combines the use of ethnographically-informed methods to establish contextual understanding of an environment with micro-analytic techniques to examine the details of interactions captured on video. It is necessarily interdisciplinary, drawing on fields such as sociolinguistics, ethnomethodology, conversation analysis, kinesics, proxemics, and ethology. The complete method presented by Jordan and Austin is intensive, involving iterative detailed study of video content by individual researchers, groups of researchers and with study participants.

4.5.3 Accounts

Errors are experienced by people and they become meaningful to others in terms of how they are talked about. Analysis of error requires an examination of the accounts people make of actions (Hammersley & Atkinson, 2007).

Examining accounts of practice has been used to characterise problems in other software engineering research. (Eisenstadt, 1993). High-level software design has been examined for characteristics of ***breakdown*** (Guindon, 1987). Ko and Chilana described applying an algorithm for ascertaining ***contention*** in open source bug report exchanges (2011). Other ethnographic studies have looked at "code talk" (Higgins, 2007) or design at the desk (Kristoffersen, 2006)

Accounts are often examined in relation to material objects. For Orr, narration was examined in relation to a malfunctioning machine (1986). In the community he observed, ***narrative*** was a tool used to fix broken photocopier machines. Keeping track of the state of machines during diagnosis was difficult, and the way technicians handled it was by ***verbally assessing situations as they developed*** and by ***providing an historic context for changes that had been made during the process***. Technicians described for each other what had been done, what these changes meant, they questioned and developed understanding, and determined the actions required to fix a machine.

Other researchers in software engineering have used narration in laboratory settings to examine conceptual design (Guindon, Krasner, & Curtis, 1987), code restructuring (Bowdidge & Griswold, 1997) or to understand how people learn to use software with tutorials (Koenemann-Belliveau, Carroll, Rosson, & Singley, 1994).

For the studies reported in Chapters 5, 6, and 7, evidence was sought of developers verbally "summing up" work (Orr, 1986). The data were drawn out of solicited and unsolicited oral accounts (Hammersley & Atkinson, 2007).

- The video recordings examined in Chapters 5 and 6 contain ***unsolicited*** accounts (Hammersley & Atkinson, 2007) generated by developers working in pairs. The videos were created for different purposes, but their relatively unstructured form permitted them to be examined for evidence of error.
- Chapter 7 drew on accounts that were ***solicited*** using semi-structured interview techniques adapted from the critical decision method (Crandall et al., 2006). To develop contextual knowledge about working environments at these sites, a day was spent observing a team at one and drew on prior-formed knowledge of the second.

4.5.3.1 Interviews and the Critical Decision Method

Interviews may be loosely or firmly structured and can be taken in different environments. They complement observation because they are social interactions (Hammersley, 2003). They can reflect an informant's desire to preserve their own reputation or to persuade the researcher to a particular point of view. Because informants are asked to reflect on their own behaviours, attitudes, character, and personality, they become reflexive collaborators in the research process (Hammersley, 2003). Eisenstadt described this well, noting that he believed his informants' accounts on the basis that he had no reason not to, and because details in the accounts were internally consistent. He concluded that accounts are sufficiently reliable if informants are given the freedom to recount experiences in their own words (Eisenstadt, 1997).

Interviewing techniques developed out of the critical incident method (Flanagan, 1954) were used to gather rich accounts of practice that would include evidence of error encounters. Flanagan's method described a set of principles to study human behaviour in relation to specific activities, or as a means to uncover the causal antecedents and critical actions taken in relation to specific events (Weatherbee, 2009). The technique has been associated

with accident analyses such as Perrow's description of "normal" accidents (Perrow, 1984) and Weick's analysis of the Tenerife air disaster (Weick, 1993).

In subsequent adaptations, the technique has been adapted for use in knowledge elicitation. For example, the critical decision method was designed to understand how people think in natural settings, developed to address the fact that the way people think in the workplace is not well explained by the findings of experimental studies of cognition. In addition to illuminating how people think on the job, the method is said to aid researchers in understanding expertise in individual domains, by revealing the differences between how experts and novices approach and manage their work. The critical decision method has itself been adapted to examine group work, and every day and critical incidents in the distant past and in the "here-and-now" (Crandall et al., 2006). The method was used in Chapter 7 to elicit focused accounts from developers about recent work, which were explored in analysis by developing timelines and narrative descriptions.

4.5.4 Incidents

Errors are encountered, they are situational. Unlike war stories (Orr, 1986) or "hairiest" bug fixes (Eisenstadt, 1997) they are often not the stuff of anecdote. They are everyday experiences, pouring-tea-into-the-tomatoes (Norman, 1981) rather than critical events that might arise in hospital emergency rooms (Crandall et al., 2006).

Error handling should be tracked ***in time*** and ***over time***. Root-cause researchers have suggested that data should not be collected too long an interval of time after events have passed, and should reflect all kinds of development activity, while researchers in psychology describe a fluctuating sense of immediacy with which the effects of error are perceived.

Variations in practice have a temporal dimension: tasks performed during the day on a hospital ward may differ if observed at night (Hammersley and Atkinson, 2007, of Zerubavel). To account for these differences, fieldwork is often undertaken by identifying

and observing "salient" periods of work and junctures, such as in periods when personnel changes occur.

In empirical studies of software development, salient periods are often defined in relation to particular tasks. Studies have examined how programmers learn to use programming environments (Ko and Myers, 2005), how they use tools to restructure code (Bowdidge and Griswold, 1997), or how they work in professional environments on set tasks such as removing a bug (Lawrance et al., 2013) or performing specified maintenance (Sillito, Murphy, & De Volder, 2008).

To meet the need of examining error within broader timeframes, incidents representing error encounters were constructed by examining accounts for verbal and visual evidence that informants perceived that something was wrong, and that they subsequently followed a process to remove effects of the error. Analytics used to identify incidents included evidence of chronological sequences (Crandall et al., 2006), shifts between progressive and evaluative problem solving (Allwood, 1984), of environmental constraints that halted work (Norman, 1981), and indicators that informants understood what was wrong and could take action to remove the effects (Reason, 1990).

- In Chapters 5 and 6, interactions were examined on video for ***indications given by developers that work had stopped***, that an error was suspected or by topics that were repeatedly discussed.
- In Chapter 7, informants were asked to ***identify a problem from recent work*** and a chronological incident was constructed out of the detailed account they provided.

4.5.4.1 Think-Aloud and Constructive Interaction

All of the videos used in studies depicted pairs of developers working together, and so understanding about how to refine representation of narrative dialogue drew on descrip-

tions and examples of protocols developed for think-aloud and constructive interaction (Miyake, 1986).

Think-aloud protocols are attractive to researchers because they provide an unbiased view of what a person is thinking while they perform a task. Verbalised thoughts give insights into how software developers reason about problems, how they shift between considering problems and solutions, and of the tactics they use to meet small goals for a larger problem (Hughes & Parkes, 2003).

The technique involves designing a task of sufficient familiarity (Crandall et al., 2006) complexity, detail and variability (Ko & Myers, 2005) to support the research question, and then collecting and recording verbalisations for analysis. In the context of software development, the technique has been used to examine problem solving in a range of different contexts. High-level design "breakdowns" that arise during set tasks have been examined for evidence of knowledge and cognitive limitations (Guindon, Krasner, & Curtis, 1987), as have the kinds of "cognitive breakdowns" made by novice users of programming systems (Ko & Myers, 2005), and studies have looked at the processes followed for recognised development tasks like debugging (Lawrance et al., 2013).

Criticisms of the technique regard the difficulties some people have in verbalising their thought processes (Hughes & Parkes, 2003), the fact that verbalising may interfere with reasoning or performance, or may be better suited for gathering information about how experts reason rather than novices (Crandall et al., 2006).

Allaying these factors, think aloud techniques may be a part of work practice in some domains (Crandall et al., 2006). ***Constructive interaction*** is a naturalistic counter-technique to think-aloud. Developing protocols out of dialogue exchanged by people working in pairs provides a view on problem-solving that is unsolicited, more naturalistic (Miyake, 1986). Unlike participants who are asked to articulate their reasoning process, pairs

undertaking problem solving tasks naturally explain to each other what they are thinking and give reasons for their ideas. Because two people are working together, a natural process of proposing, testing and defending ideas is made available for analysis (Miyake, 1986).

The method shares common points with think-aloud protocol. Studies give participants a set task which they are to solve together. In Miyake's study, participants were asked to "figure out how a sewing machine makes its stitches." (Miyake, 1986, p. p.159). Six participants with varying degrees of prior experience with the machine were assigned to one of three pairs and worked together in sessions that were video and audio recorded. Each pair undertook three sessions during which they solved the problem using different tools: pen and paper, the machine, and a machine with thread. Miyake analysed the statements made by participants during each session, which she mapped to one possible explanation of how the sewing machine creates a stitch.

Constructive interaction has been used in software engineering for human computer interaction research (Wildman, 1995) and to study collaboration and team work (Flor, 1998; Flor & Hutchins, 1991) Bowdidge and Griswold used the technique to study how programmers restructure code using different tools (1997). They noted that one of the strengths of the technique is that it can be moved out of the laboratory and to the desk, and thus into a familiar environment that may yield dialogue and actions on the computer that "reflect habits and patterns typical of the programming culture" (1997, p. 230).

Though they do not specifically cite constructive interaction as a methodological orientation, studies that examine pair programming benefit from access to naturalistic exchanges of dialogue. Dialog-based verbalisation is necessary during pair programming (Xu & Rajlich, 2005) and the activity has been studied for attributes like attention (Sillitti, Succi, & Vlasenko, 2012), and engagement (Plonka, Sharp, & van der Linden. 2012).

4.6 Summary

This chapter described the methods used to perform this research. First, the commitment to using ethnographic principles was described. The aims of the research were given and an overview of specific methods used for gathering and analysing source material was provided. Finally, an introduction was given to the field sites and the people who informed this work.

In the next three chapters, reports are given for studies introduced in Section 4.4 that examine practice from different perspectives. The studies draw on data gathered using different methods and have been examined using media and format specific analytic techniques. Interpretation has grown out of employing principles of thematic analysis, but the texts and analyses are also structured temporally (Hammersley & Atkinson, 2007, p. 195), according to the broad sequence of activities that have been identified in psychology research as being a part of the error handling process. Two aims for reporting emerged from analysis:

- First, to establish a representative ***catalogue of error handling features*** detailing encounters reported by developers working in different settings and on different tasks.
- Second, to develop the ***descriptive framework for error*** used in software engineering by representing individual encounters within technical, social, and organisational contexts.

5. At the Drawing Board

Design is a prevalent, central concern in software engineering, comprising both the goals to be achieved and the means: the particular tools, materials and mechanisms employed to meet them (Taylor & van der Hoek, 2007). As in other disciplines such as architecture, the need to continuously comprehend and compose permeate all of the tasks undertaken in software development: domain problem understanding and representation, the development of corresponding technical specifications or plans, writing the code and maintaining it. Performance of these subtasks alternates over the course of an initiative, interleaved and interwoven by the basic processes of understanding and construction (Pennington & Grabowski, 1990).

Taking this perspective, it is possible to consider that features of comprehension and composition from any one area of software development may resonate or have relevance when examining other tasks. Findings about design practice can be used to frame and lend context to examinations of development at the desk, or to stories of recent work gathered from organisational settings.

Design practice is commonly examined as it is performed (Cross, 2001). One aim of the study reported in this chapter is to orient descriptions of error encounters and error handling to prospective analytic techniques that are established in software engineering research. A second aim is to distinguish aspects of problem solving in software development activity that are performed in-the-moment, that are local, tactical (Reason, 1990) or reactive (Eraut, 1994) from other activities that are strategic (Reason, 1990) or deliberative (Eraut, 1994).

The following pages examine the ways in which developers manage difficulties that arise in a paired design session. The chapter begins with a review of concepts from related

design studies, followed by a review of concepts and theories related to problem solving from human error research. The scene is set, followed by findings and a brief discussion.

5.1 Related Work

Prior findings have described the kinds of ***breakdowns*** that designers encounter (Guindon, Krasner & Curtis, 1987). Software design activities are hampered by three kinds of breakdown. Designers may lack specialized computing knowledge or domain knowledge. They may also experience failures of memory or possess inadequate tools to support reasoning. Some breakdowns are blends of the first two: characterised by aspects of knowledge and aspects of cognition.

Software designers use specialized knowledge when performing early design tasks (Guindon, 1990). They retrieve or simulate scenarios about the problem, elaborate requirements, identify inferred constraints and discover new requirements. Solutions are developed and represented using external representations. Designers use heuristics to assist solution generation, finding ways to simplify tasks, by delaying commitment and otherwise reducing the complexity in order to avoid making serious mistakes.

These attributes have also been described in the more general context of design cognition. Cross identified three characteristics of design work that provide simple framing principles for analysis of design activity. ***Problems are ill-formed,*** identified in tandem with solution generation. ***Solutions are opportunistic***, following a realistic strategy of finding a "satisfactory" rather than "optimal" solution. ***Process is ad-hoc and unsystematic***; designers are wary of process that has not proven itself (Cross, 2001).

These points also resonate with descriptions of problem solving in the context of error handling. Strategic problem solving, like much of design work, is future facing, linked to goals that are ill-formed, dynamic, and which can only be assessed after time has passed (Reason, 1990). Solutions must, of necessity, therefore be opportunistic, satisfactory rather

than optimal or correct. Design practice is also tactical, particularly at moments of difficulty. Local problem solving will be used in these cases to work through difficulties. Designers will draw on tools such as generation of alternatives, and sketching as they identify local problems that can be solved and assessed in order to return the focus of work to the larger, strategic task.

Other papers have analysed the same data as the study reported in this chapter. The studies are notable because they attend to aspects of ***awareness*** that arise during design. Designers must be comfortable with a degree of uncertainty and ambiguity in order to create (Cross, 2001). Phenomena of uncertainty such as vagueness, hesitation and delay serve design process by making collaboration possible (McDonnell, 2012).

Within the framework of error handling, these phenomena would be taken as indicators of suspicion, the sense that something is wrong in work that was previously completed (Allwood, 1984). They might also be taken as indicators of turbulence, that the designers have lost or are in danger of losing control of the process (Amalberti, 2001). If evidence of either factor remains at the close of a session, such as in comments indicating ongoing dissatisfaction or aborted problem solving, one might also surmise that the issue remains, in some sense, active.

In *Software Design as Subject- Oriented Design Cycles*, Baker and Hoek examined the development of ideas in software design, looking at evidence of strategies and patterns used by designers in idea generation, evaluation and acceptance. Their method identified cycles within design, periods of time delimited by moments of focus-setting. As in the findings presented here, a high incidence of ***question asking*** was observed within design sessions. The study took such activity to be evidence of *uncertainty* and a *lack of creative forward movement* (Baker & van der Hoek, n.d.)

Nickerson and Yu examined the nature of the collaboration at moments of ***conflict***, and included in their analyses examination of speech as well as of other conversational activities such as gesture and diagram (2010). Their findings suggested that conflicts arise because designers attach themselves to *divergent perspectives* that meet the requirements of individually selected evaluation criteria.

McDonnell examined how designers in the SPSD situations use verbal interaction to explore the mechanisms designers use to keep a design process moving in spite of disagreements. One tactic designers use is ***tentativeness***, employed to simplify a task, or to set aside issues that will be considered elsewhere. ***Disagreement*** is accommodated through the use of *indicators*, including the use of hypotheticals, by accommodating conflicting ideas in the design process, either by relating both possibilities to the larger design, or by using distinct terms to set the solutions apart.

5.2 Setting the Scene

The design session analysed within the former studies and in the study reported here was collected as a part of the NSF funded International workshop "Studying Professional Software Design" (SPSD), held February 8th-10th, 2010, at the University of California, Irvine. The goal of this workshop was to collect observations and insights into software design, that could be related to theories and methods from a variety of research disciplines including software engineering, design studies, human-computer interaction, cognitive science and psychology.

Workshop participants analyzed a common set of data comprised of videos and transcripts of three paired interactions of professional software designers. Each recorded session lasted for approximately two hours. The analysis given in this chapter examined one of the three sessions, commonly referred to in other studies as the ***AmberPoint***

Session. More information about the workshop may be found at: http://www.ics.uci.edu/design-workshop/ (http://www.ics.uci.edu/design-workshop/).

The next section introduces Kasia and Bill, and describes the setting for their design session. This chapter does not include a full account of the methods that were used for collection and analysis. For this information, see Chapter 4, Section 4.3.3. It may also be helpful to consult Appendices A and B.

5.2.1 The Amberpoint Session (Site A)

The Amberpoint Session depicts design activity performed by Kasia and Bill, identified in Table 5.1, below. The pair are experienced designers, and are colleagues at an industrial firm. Kasia and Bill were given a design prompt specifying high level requirements for a traffic flow simulation program (see Appendix B.2), and were asked to produce a conceptual design for the system. The pair were asked to record design decisions on a whiteboard.

Site	Name	Gender and Age	Experience
Site A	Bill	Male, thirties	Professional Designer
	Kasia	Female, thirties	Professional Designer

Table 5.1: Informant demographics, Site A.

The session lasted for one hour and fifty-three minutes. It was filmed using two fixed cameras placed at different angles and proximity to a whiteboard. The session began with Kasia and Bill sitting at a table, reviewing the design prompt. Following this, the pair move between sitting at the desk and standing at the whiteboard. Bill does most of the diagramming and note-taking at the whiteboard, though Kasia stands at the board during discussion to reference and consult diagrams. The final six minutes of the film depict reflection and review of the session.

5.3 Findings

In this section, three incidents are analysed. Findings are characterised in terms of concepts drawn from Guindon's kinds of knowledge (1990).

5.3.1 I don't know if I like the pop-up window anymore.

Kasia and Bill work through the design of traffic signal timing by diagramming how it will be represented in a user interface. The difficulty unfolds over three segments, with two additional minor mentions made to it: one roughly three quarters of the way through the session and one within the reflection period. The second segment is the longest of the three, lasting approximately fourteen minutes.

In the course of specifying behaviour, the interface component undergoes several iterations, depicted in Figure 5.1, below. Bill works predominantly in the solution space, as indicated by what he draws: how he extends, alters or removes bits of screens on the whiteboard. Kasia works within the problem space. She verbally explores aspects of the problem, and uses design heuristics (Guindon, 1990) to simplify the problem.

In the second segment, Bill does not accept Kasia's suggestion, that the problem be simplified ("Kasia: So, so you don't have to specify all four, because you only need to specify one or two and the other ones are implied because, you know--"). He pursues instead an attachment to developing the timing solution visually (#00:35:50.0# "Bill: --I understand what you mean, I understand what you mean but I think part of the traffic light problem is figuring out how long we should have the overlapping red lights").

The resolution is opportunistic: the third partial solution that is generated is accepted as sufficient. The resolution is signalled by the invocation of an external constraint, and mention is made that more detailed design work will be required. However, no additional work is done during the session, particularly within the period during which the primary representation, entity relationship diagram is developed.

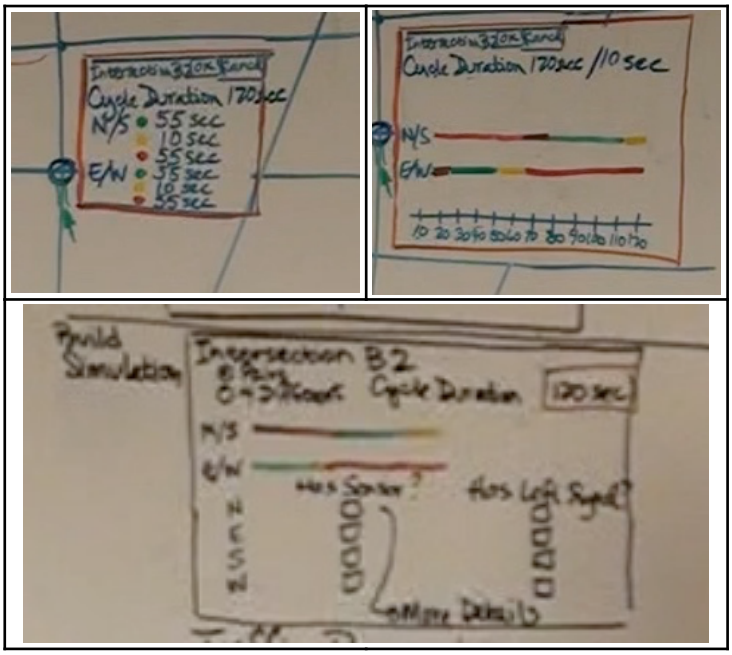


Figure 5.1: User interface representations of traffic signal timings.

In the traffic signal timing issue, the resolution is incrementally represented in sketches, perhaps firmly enough to indicate that the problem is resolved. However, the representation is uneven. Bill's attachment to a user interface solution and the subsequent lack of development of the underlying object model could signal that the issue is still active. This possibility is hinted at by Bill's ongoing suspicion, indicated at the end of the session that he is not satisfied with the solution produced (#00:40:43.0# M: "I'm still--the input I'm still unhappy with the light timing").

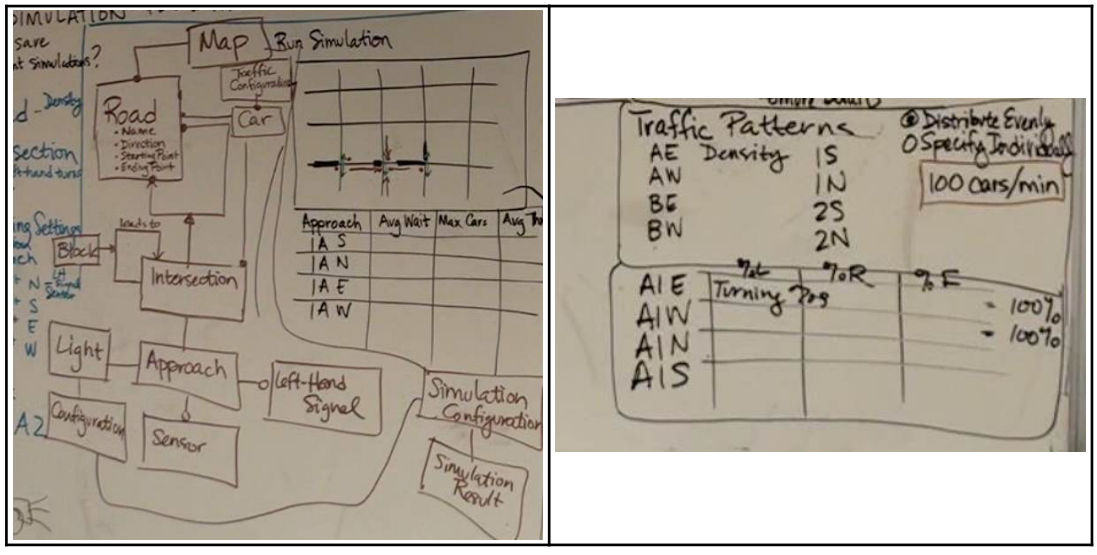


Figure 5.2: Traffic signals. The entity diagram includes elements for cars and for intersections, as well as for managing traffic. The diagram on the right is an element of the user interface, indicating how traffic patterns might be configured within a simulation.

5.3.2 So you think there should be a car out there?

Kasia thinks that cars have a distinct identity within the system and must be modelled while Bill believes that cars are handled more generally as a part of how traffic passes through intersections. This difficulty has four segments; the second marks the point at which the difficulty is named, while the resolution comes in the third.

McDonnell noted that the adoption of different terms by designers accommodates disagreement, but it may also signal more fundamental ill-formed understanding, a planning barrier (Frese & Zapf, 1994). It may be that the next action to take is unclear because the intention cannot (yet) be named.

Conceptual consensus is achieved through discussion. The pair make extensive use of scenario solutions. Two domains figure heavily in this process: that of the way traffic works, and of how simulations should behave for users. Gesture is used by both designers in the course of these scenarios to convey level of agreement and to express ideas. At times, consideration of the problem is constrained by references made to external constraints (#00:13:20.5# "Bill:... Professor E must have statistics").

The solution is partial. It takes two representations on the whiteboard, depicted in Figure 5.2. One is in a low-fidelity representation of a user interface component. The representation is of a lower fidelity than the one developed for the traffic signal interface: no colour is used, a second part of the diagram is tacked on to the first, and numerous abbreviations are given to indicate fields on the screen. The solution is also represented within the entity relationship diagram showing how major objects within the system relate to one another.

Both designers seem to be in agreement at the end of the session that traffic patterns needed to be configured and managed independently of intersections and of cars, and this is represented in the diagrams. This may mean that the design difficulty has been suffi-

ciently resolved to permit unambiguous action going forward. However, even in the last moments of discussion surrounding the issue Bill asserts that the intersections should have some control or knowledge over the way traffic patterns are managed, suggesting that he remains unsatisfied ("#00:33:15.8# M: But, it might ask the approach what the traffic configuration is.").

The desire to draw on the support of problem solving at the desk is a marker of contingent recovery. Kasia and Bill mention that a number of the requirements for timing might be worked out through implementation, a known strategy in development (LaToza & Myers, 2010).

5.3.3 Ultimately, you want to know whether it worked.

Kasia wonders how the success in performing a simulation using the software is to be determined. She argues that success relates to how factors such as speed, distance, and car density on roads should optimally be combined by students to produce simulations. The difficulty is discussed in three different segments, the first occurring early in the session, and the last forming a substantial part of the reflection period at the end of active design.

Unlike the previous difficulties, the "working" issue is primarily discussed in relation to other parts of the system, such as in relation to the creation of a summary area or dashboard for showing how the simulation is configured (segment 1), or the effects of running a simulation (segment 2). This means that over the course of the session, very little is captured about the problem except as it might relate to partial solutions of these related issues.

Turbulence during design work is indicated with questions, gesture and in terminology that is fluid, changing. It may also be indicated by repeated discussion about a specific topic. In these cases, representation may take the form of little information captured in the

representation, or of unbalanced capture wherein part of the design is highly detailed, while other parts are not.

In fact, at the end of the session, the entire discussion is only represented on the whiteboard via additions to the listing of requirements maintained by the designers: one question, and three unlabelled references to attributes for intersection approaches as shown in Figure 5.3.

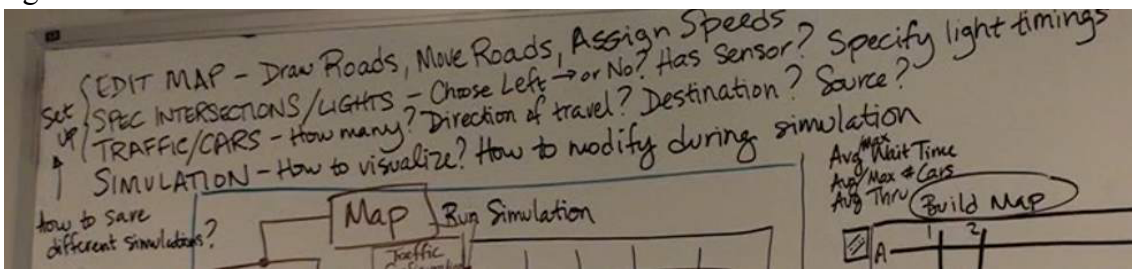


Figure 5.3: You want to know it worked. The discussion related to this issue is not represented in diagrams, but in a list of requirements and one question indicated in black. "How to save different simulations?" is noted on the left as are the notes “Avg/Max Wait Time", "Avg/Max # of Cars", and "Avg Thru" to indicate the average throughput to be expected for each intersection per minute.

This is the only difficulty examined that is discussed at length by the designers in the reflection section, during which they indicate that they didn't feel satisfied with what they were able to achieve. The issue is left unresolved, though the designers suggest that the next steps would be to go back to the Professor, and to explore the problem by virtue of implementing parts of the system that were more firmly captured in the design representations.

5.4 Discussion

The Amberpoint session has been characterised as an example of expert design. Kasia and Bill move between contexts: those of design and use, depth and breadth, and between the requirements and the design (McDonnell, 2012). The session reflects the general features of design cognition. Evidence is given by the designers that they are engaged in problem formulation, solution development, and process strategy (Cross, 2001). The session

demonstrates instances of problem framing, co-evolution of problem and solution, attachment to concepts and modal shifts.

There is also evidence for the more detailed kinds of knowledge observed by Guindon in the context of software design. For example, there are many examples of requirement inference, the clear emergence of preferred evaluation criteria, and a number of design heuristics are used, such as considering a simpler problem first, and delaying commitment.

The incidents that were selected were distinctive for several reasons. Topics were repeatedly discussed. The designers re-examined tentative decisions (Guindon, 1990), or exhibited attachment to concepts (Cross, 2001). Exchanges include language that indicates disagreement (e. g. "I don't think so") or lack of understanding (e. g "I don't know"). The designers indicated they were doubtful, or that they were not confident in the ideas being expressed. These moments were signalled by repeated turns away from the whiteboard and corresponding provisions of assent in the form of paralinguistic utterances (e. g. "mm hmm", "yeah"). Episodes also exhibited evidence that the concepts under discussion were difficult to represent, as indicated through repeated use of problem framing in exchanges, references to the design prompt, or extensive re-working of diagrams.

5.4.1 Scenarios

Guindon found that designers rely on mental simulations of the solution space to evaluate the in-progress solution: to determine how complete it is, and to tease out any "bugs" or inconsistencies it contains. She states that "solution simulations were done in terms of test cases based on problem domain knowledge" in her case of possible scenarios for how lifts should behave (1990, p. 291)

Kasia and Bill appear to use the scenarios to explore two problem domains: that of how traffic works in the real world, and of how users interact in general terms with simulation

software. Simulations in this session often explicitly refer to the *use* of the system, particularly in regard to how other simulation user interfaces are known to work, and the bearing that this knowledge may have on the current design effort. This may simply reflect the emergence of user experience as a preferred evaluation criterion for this session. It may also reveal lack of experience in solving this kind of design problem.

Software development must solve problems both within a problem domain and within software engineering (Pennington & Grabowski, 1990). Within safety science, this has been described in the context of "object worlds": the different domains to which an object of design belongs. Part of the task in these cases is to determine what constitutes acceptable conditions in each domain for the effects of decisions that are taken (Rasmussen, Pejtersen, & Schmidt, 1990). Because of this, it is reasonable to assume that analogous scenarios may. of necessity, draw from both sources.

5.4.2 Constraints

The design prompt was heavily used by designers in the Amberpoint session to select terminology, to check requirements, and as a means to evaluate the completeness of partial solutions. These uses may indicate evidence of *requirements elaboration*, defined by Guindon as being performed to reduce ambiguity in the design prompt and to decrease the field of possible solutions (Guindon, 1990, p. 290)

There is some evidence that the prompt is used during exchanges to express uncertainty ("Did it say that?", "...or did I read something there that said it has some"), or to signal disagreement ("Potentially, I don't know if it needs to be that complicated but I could see"). The purpose of the prompt does not always appear to serve as a direct source of information, but rather as a marker of something outside the design exchange. It is in picking up the prompt that a message is conveyed, that conversation is diverted or paused.

More explicit boundaries are set around problem solving by making reference to the past and to the future. These references locate the problem solving activity in the present, in the now. They set the current moment apart from decisions that were taken before (e.g. "[W]e have to talk to Professor E again") and those that will have to be taken later ("That's version 2.0").

Boundaries set by lack of understanding may signal a difficulty that will occur in later activities. Indications may take the form of explicit reflections that something is not understood, naming the difficulty, use of the requirements specification for problem framing, or by constraining responsibility by making references to the client ("We need to go back to Professor E").

5.4.3 Representations

Sketching has been named by Cross to be the “primary” tool that supports design cognition, facilitating the “the uncertain, ambiguous and exploratory” aspects of design. It assists designers in generating tentative solutions, identifying what is still not known and revealing emergent properties and features (Cross, 2001, p. 17).

In addition to sketching, Kasia and Bill employ gesture and use questions to structure and frame problems. These devices do not stand alone, but are used in reference to an immediate aim, such as in uses of the design prompt to elaborate requirements, or as a part of solution simulation.

5.4.3.1 Sketching

Kasia and Bill were given the requirement that they use the whiteboard to record decisions, and so the session includes extensive sketching and listing. Different kinds of information are depicted using different colours. The main simulation interface is depicted in blue, while lists of requirements are always recorded in black, bulleted points. The object model

is sketched in brown. As one would expect, the user interface elements that correspond to traffic signal lights make use of red, green and yellow.

Colour selection is purposeful within the representations, but may also indicate a preferred working practice. Most of the sketching at the board is done by Bill, an arrangement that appears to be comfortable to both members of the pair. At one point, Bill uses humour-infused blame to suggest that someone has "taken" a colour away from him, as depicted in the exchange below. This comment is made in jest. Kasia laughs in response, and the sense is given that this is how they normally work together.

(00:39:43.5)

***Bill**:--If you wanted to. And then we could have something, we could have another color, we could have another color that represents green arrow if we need to. Left, left−turn is orange, I don't know.*

***Kasia**: Purple.*

***Bill**: Purple?*

***Kasia**: (Do you have a purple?)*

***Bill**: Who took purple away from me (inaudible)?*

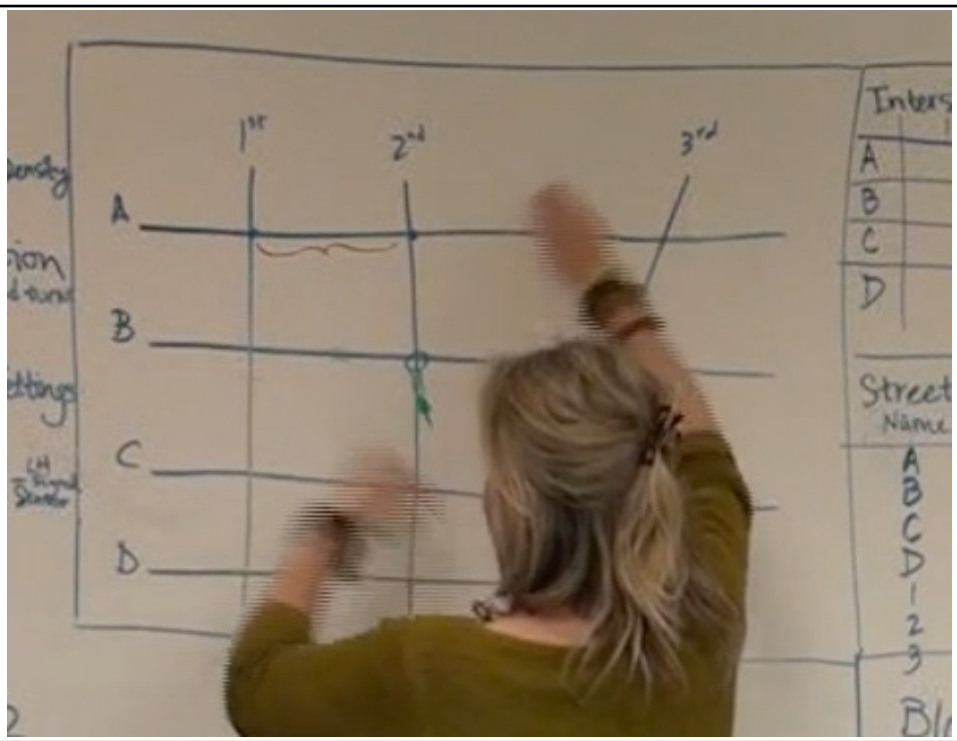
***Kasia**: (Laughs.)*

***Bill**: Brown, how 'bout brown?*

***Kasia**: Okay.*

5.4.3.2 Gesture

Bill and Kasia perform gestures over diagrams, in the air, or over a physical object, such as a table. Guindon does not mention the use of gesture in her analysis of software design, but it reasonable to group this device with other uses of external representation.



00:20:02.7
Kasia: --So we assume they have some sort of a pallet here where they grab a road and they start dragging [F makes motions with both over the diagram]
Bill: Yeah, that's true.
Kasia: and then they drag that, so there is some sort of drawing pallet right, that says okay I have this thing I drag something, I'm drawing a road and I call it something and I draw and I call it B and I draw my roads and then--

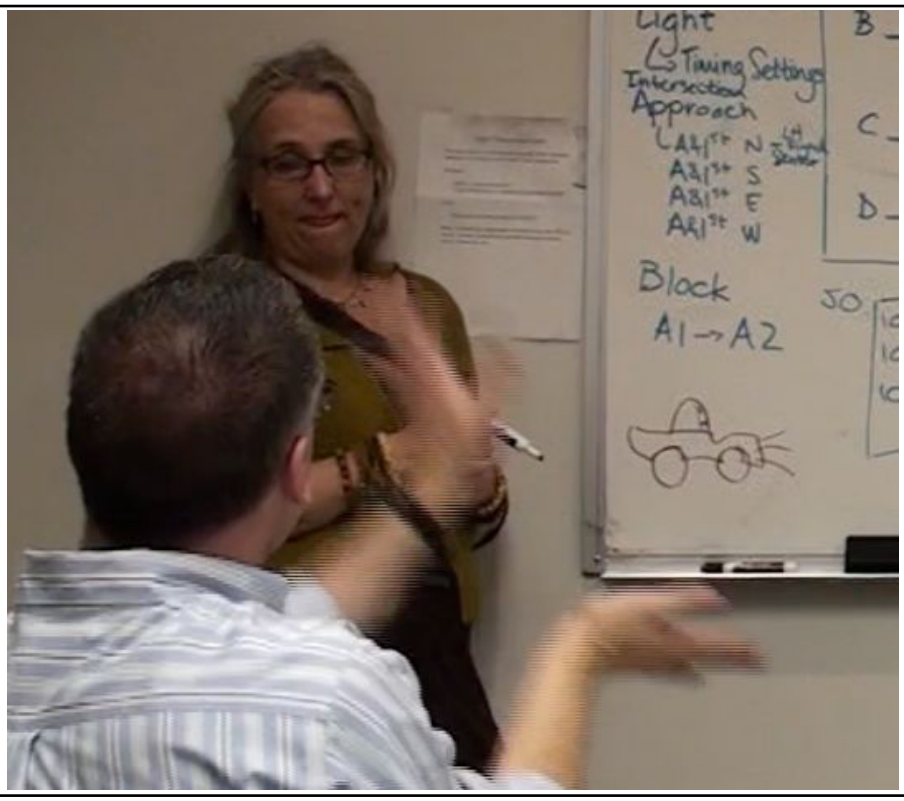
Figure 5.4: Gesture invoked to model traffic signal timing (Section 5.5.1). In this example, Kasia uses her hands to indicate a road being placed into the grid space. Her hands are roads, the diagram on the whiteboard represents the simulation. The gesture is low-commitment: Kasia can use the gesture without altering the diagram that is represented on the board.

Gestures provide reference points to discussion (e.g. "if you click on here and double click on here") (Nickerson & Yu, 2010). They do this by providing common objects that serve discussion. They are unambiguous and commitment free: it is easy for individual designers to make or to replicate a simple gesture over a diagram. Nothing needs to be committed to the design in this way, nothing needs to be altered or removed. An example of this is shown in Figure 5.4, above.

#00:56:32.2#
Bill: ...and usually stay the speed limit then they usually stay and go green throughout and so then, then on a given street, [Bill makes motions on table with hands] all the lights have to be timed in such a way that if you follow that, if you follow that speed limit then you will get to the next green light and you'll never have to stop as long as you stay at the speed limit.

Figure 5.5: Gesture invoked to model the problem domain. In this example, Bill uses the table to represent streets, and his hands are cars. The gesture is communicative, but also knowledge bearing, representing how cars are known operate in the world.

The same gesture can be used to serve discussion about the problem and solution space. Gestures are often knowledge bearing, conveying how something behaves in the world or how it is intended to operate within software that is being built. An example of this is given in Figure 5.5, above. Gestures may be used to inform but also to persuade, invoked to punctuate the idea that is being conveyed with a visual example. An example of this is given in Figure 5.6, below.



00:46:25.6
Bill:--No, no, then I think you would say um between these hours, 50 cars start at B and then
Kasia: mm hm
Bill: given the percentages, they're going to turn at--
[Bill holds hands together and then moves them apart.]
Kasia: --mm hmm
Bill: 9am.
Kasia: So that's what I was getting at, so you need direction, south.

Figure 5.6: Gesture used to align understanding in the cars incident. In this case, Bill illustrates the principle of turning with his hands to support an argument for managing timing. Kasia responds by discussing directions.

5.4.3.3 Questions

A high incidence of questions was observed to be asked by designers in two contexts. As a part of problem framing, Kasia and Bill use questions to introduce different (but ultimately a defined set of) terms to articulate concern about the problem they are trying to solve. As a part of solution framing, questions are asked to clarify understanding in situations in which one designer has put forth an idea. Guindon does not cite question asking as a

significant feature of either activity. The sense given is that this activity may relate more to the attributes of discussion than to the use of particular kinds of knowledge.

Questions may also indicate that the designers are becoming aware of an error, or mark shifts from prospective to evaluative problem solving (Allwood, 1984). Suspicion indicated through questions, as do tentativeness or hesitation, signal that something is not right. They are part of the emergent, contingent process of assessing what works through the detection of things that ***don't*** work (Alexander, 1964), but also can serve as markers of reasoning, shifting frames of reference that accompany error handling (Rizzo, Ferrante & Bagnara, 1995).

Other modulators that are often framed as questions such as doubt or worry could also be grouped as a representative tool that supports design work. In all three of the incidents, there is a moment when the designers articulate what they perceive to be the crux of the design difficulty. In the context of the "So what does 'It worked' mean" episodes, it is the high frequency of questions that indicates a problem. In the other episodes, declarations and banter are used ("Gosh, who knew this was so complex?", "I know, it's amazing") , through an admission that a miss-communication has occurred ("You see, I thought-"), or that something has gone wrong ("I feel").

5.4.4 Limitations

The data were examined in recognition of several limitations. Though the film depicts naturalistic exchange between professional colleagues, the session was laboratory-based and the informants were given a set design task. The sources were secondary (McGinn, 2008), it was not possible to develop a full understanding of the professional context for the session: the environment in which the designers normally work, their individual backgrounds or preferred working styles, or of the kinds of design problems they normally solve.

Partial understanding of industrial design was developed by comparing findings in this data with a second set of films that depict "real world" design activity in an industrial setting. Those films depicted design of a "real world" problem in an organisational context. Though informative, the second set of films was not rigorously analysed, and findings have not been used in reporting.

5.5 Conclusion

This chapter reported a qualitative study undertaken to examine how designers work through difficulties encountered in high-level design. It examined the activities of two designers working at a whiteboard on a set design task. The analysis demonstrates the applicability of principles from design cognition to collaborative laboratory sessions. It provided evidence of active qualities of problem solving in deliberative activities like design that align with factors of problem solving characterised as being used during error handling.

Agreement and shared understanding are not necessarily indicators of resolution. They may be indicators, however, of undoing effects within a local ***recovery***. As McDonnell noted, strategies like vagueness, hesitation and tentativeness support collaboration and allow work to move along. Indicators of suspicion may also signal that recovery is ***provisional***.

One way that designers resolve issues is to set boundaries (Guindon, 1990). Kasia and Bill state at points that that the next step to take would be to return to the client for clarification of the requirements. By admitting a lack of knowledge, designers set a boundary on their responsibility, but they also indicate an awareness that the issue remains active, that it may require additional handling. Likewise, designers can recover from difficulties by banking on future work: through additional discussion with stakeholders,

but also work at the desk. The best indicator in this case is how designers discuss future activities like prototyping in the context of current activity.

Recovery can be roughly assessed by how steady the process is, and how by how satisfied designers are with results. A process that leaves the designers ***satisfied*** is marked by the adoption of unambiguous terminology, and by the degree of capture within and balance between individual design representations. Within a ***turbulent process***, developers may not be able to represent a solution, but only to frame the problem; discussion of the solution space may primarily reveal inconsistencies and gaps within the requirements.

6. At the Desk

Problem solving in software development is often strategic: success will be determined only after time has passed and the outcomes of decision making can be assessed. The strategic element is particularly strong in software design, in which plans and preparations are made for other software development tasks. Decisions taken at the whiteboard that result in clear representations in diagrams or verbal agreements suggest resolution of immediate concerns. However, setting constraints on problem solving and other indicators such as feelings of unease or dissatisfaction may signal that problems remain active. Resolution may be achieved only on the promise of work that will performed at a later time.

At the desk, developers must continually interpret what has been recorded within software that they use and write. Distinctions that were previously made must be examined, and new commitments must be made within a process that is never-ending and never-complete (Winograd & Flores, 1987, p. 73). Software is social and historic, a medium that reflects the intentions of other developers and of individual developers at particular moments in time.

The aim of this chapter is to examine how such distinctions are made by developers, how comprehension and composition is undertaken at the desk (Pennington & Grabowski, 1990). The focus is on moments when "misfits" between developers and machines (Rasmussen, 1985) or "break-downs" (Winograd & Flores, 1987) in action arise.

The chapter first reviews studies that have examined problem solving in software development, and concepts and theories related to problem solving from human error research. The project depicted in video recordings made by two open-source developers is introduced. Following this, an analysis is made features that were observed in different

incidents, culminating in a discussion of activity at the desk in the context of error handling.

6.1 Related Work

Traces of decision making undertaken at the desk have long been kept and reported in tools and mechanisms designed for other purposes. Bug trackers are used to keep track of information about "almost bugs" (Bertram, Voida, Greenberg, & Walker, 2010). Comments are used to track information about work that is yet to be completed (Storey, Ryall, Bull, Myers, & Singer, 2008), bugs are reassigned so that those known to have active experience with issues can see them through (Guo, Zimmermann, Nagappan, & Murphy, 2011). Reports of error often drive and organise practice, but have also been shown to be incomplete and inaccurate, with gaps of information that must be filled through interactions between developers (Aranda & Venolia, 2009).

Social interaction is considered to be a cornerstone of writing good software. Hoare argued that a culture of reliability would result over time in the "natural" emergence of dependable software (1996). Weinburg famously described this as "ego-less programming", environments in which shared, open access to software replaced older cultural values of programming as a secretive, solitary practice. Ego-less groups increase awareness of what is in code, and in so doing facilitate error detection during writing (1998).

Study of interactions in formal meetings and in work at the desk support these ideas. Study of code inspections has shown that the length of meeting times and the physical proximity between developers can influence the number of defects that are reported (Seaman & Basili, 1997) Examinations of work at the desk argue that software is social "to the core", that meanings within software arise out of the interactions developers (Higgins, 2007).

Interaction has long been argued to make programmers “better” (Weinberg, 1998), however, less is known about how individual developers make decisions. Studies have examined the information needs of developers (Ko, DeLine, & Venolia, 2007), how they “forage” for information during bug-fixing (Lawrance et al., 2013) and the personal strategies they utilise more generally during development (LaToza & Myers, 2010).

The argument has been made that decision making at the desk is like problem solving associated with design. Design work cannot be moved "upstream" from programming, because it is a constitutive element of programming. (Kristofersson, 2006). The names given to elements of software become the design and the design depends on what things are called. Some errors at the desk are handled by designing them away, making them acceptable by accounting for them. Likewise, renaming pieces of code can make them "right", so that they fit new, emerging requirements (Kristofersson, 2006).

Collaboration and coordination studies examine the environment in which software is created and the ways that tools and process support the coordination of activities. They also explore how work is mediated by talk and by the records associated with software development: bug databases, code repositories, and in some cases, source code. As in the root-cause analyses, the studies primarily consider error in terms of outcomes, and examine most closely activities like bugfixing that have long been associated with error detection and removal.

6.2 Setting the Scene

This section sets the scene for the findings reported in Section 6.3. The videos used in analysis were created by Marcus and Joe, two active members of the professional agile community. The following sections introduces the project, developers, and provides an overview to how practice is organised.

The section does not include a full account of the methods that were used for data collection and analysis. For this information, consult Chapter 4, Section 4.3.3 and appendices A and C.

6.2.1 Acceptance Test Framework (Site C)

This project is extending an open-source acceptance test framework to allow users to specificy "literate" acceptance tests. Wiki-based and written in JAVA, the framework was designed to allow non-technical users to specify and run acceptance tests for software. The altered project will permit tests to be written that follow the Given-Then-When pattern (North, n. d.). Classes and packages will be named so that they can be parsed and presented to readers on webpages in a form that approximates natural language. Likewise, users of the framework will be able to create tests with names that are readable and easy to understand.

Development draws upon a JAVA interface written by Marcus some months prior to filming. That project had two aims: to test out ideas about writing human object oriented application programming interfaces (API), and to support the separation of roles and tasks in behaviour driven development frameworks. It included examples which demonstrated the use of the API in conjunction with the open source acceptance test framework that is being altered by the pair. The API is used directly at points, and examples included in its documentation are referred to and borrowed from.

6.2.1.1 Informants: Marcus and Joe

Marcus and Joe, identified in Table 6.1 below, perform all programming tasks together. The two are more or less evenly paired, each has been programming professionally for over ten years. Both programmers are familiar with the acceptance test framework, however Marcus appears to have more recent experience in developing within it.

Site	Name	Gender and Age	Experience
Site C	Joe	Male, thirties	Professional Consultant, 10 years
	Marcus	Male, thirties	Professional consultant, 10 years

Table 6.1: Informant demographics, Site C.

By contrast, Joe exhibits greater familiarity with the tools that are being used, in particular with the IDE and a continuous unit testing plugin for the IDE. Evidence is given that he takes the lead on performing upgrades on these tools between filming. There is also evidence to suggest that he is an advocate for using Linux or Unix variant operating systems, and that his most recent development work has been done in Cocoa.

Watchers

The episodes that were analysed were filmed in office environments, as depicted in Figure 6.1, and there are frequently people co-located in the room where development is happening. Watchers predominantly follow along with the programming action, but also comment from time to time. At times, their input affects the work. Watchers differ between episodes; no single Watcher is consistently present.

Episodes were webcast using web meeting software and Watchers also participate via the internet. People drop in and out of the sessions, at times commenting or asking questions in real-time via chat. When this happens, a co-located Watcher brings the question to the attention of Marcus and Joe, or one of the developers notices that a question has been asked in real-time chat. In both cases, the question is addressed in the course of ongoing work.

6.2.1.2 How Practice is Organised

Marcus and Joe use the Eclipse integrated development environment (IDE), and create extensions to the wiki-based acceptance test framework. They also use the wiki to direct

their work, writing stories within it that define the functionality they intend to add to the framework. The wiki environment is viewed in Firefox.

Figure 6.1: Development sessions were held in offices.

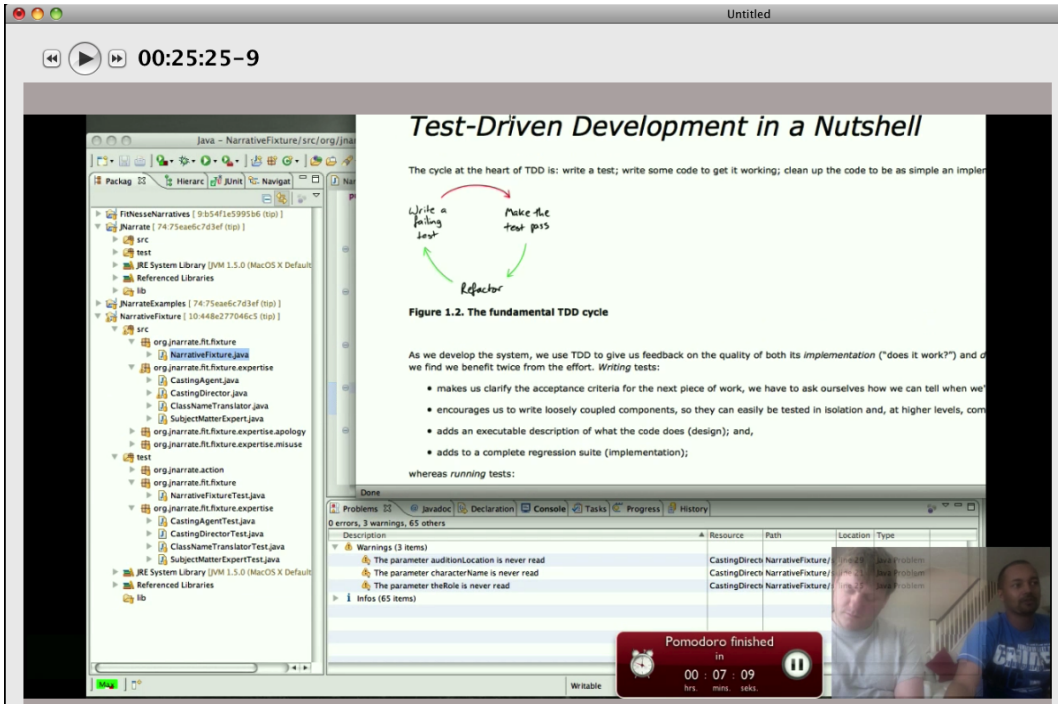


Figure 6.2: Filming depicted a screencast. After episode 20, the screencast included video of the developers at work, prior to that, only the IDE or web browser output was visible. This figure also depicts a screen explaining test-driven development principles followed by the developers, and displays the “pomodoro” timer Marcus and Joe used to keep track of episodes.

The developers take turns driving and navigating. One writes a unit test, defining proposed behaviour for a class, and the other implements the behaviour by adding necessary methods to the class. The driver often informally "thinks aloud" to indicate the actions he is taking. Likewise, the navigator often acts as narrator for the audience, explaining in broader terms what is being done, and how it is oriented within the larger goals for the project. In addition, the two interact with each other, discussing the work that is being performed.

Development is undertaken on a Windows laptop owned by Marcus (see Figure 6.2 for a representative image of the screencast depicted in video recordings). In the first and subsequent episodes that were analysed, the performance of this laptop distracts the developers and slows progress.

6.3 Findings

As described in Chapter 3, Section 3.3, error handling is generally described as a three-stage process (Brodbeck, Zapf, Prümper, & Frese, 1993). A person must know that an error has occurred, identify both what was "done wrong" and "what should have been done" and then understand how to "undo" the effects of the error (Sellen, 1994, p. 476). Handling unfolds in the course of "progressive" problem solving. An error is suspected or detected, and an evaluation is made to identify the source of the problem (Allwood, 1984). Environmental cues supply feedback to the problem solver by blocking forward progress (Norman, 1981), communicating about problems in system state (Lewis & Norman, 1986) or by circumstantially guiding a problem solver to recovery (Reason, 1990).

Following this rubric, features of error are illustrated in the findings that follow using statements and exchanges of dialogue drawn from error handling incidents. Forty-three incidents were examined in detail; a diagram depicting the spread of incidents across episodes can be seen in Figure 4.4. Taken as a group, incidents are notable for the time

they span: many are fewer than five minutes in length with clearly defined points of detection and resolution. Other handling processes took much longer, with the longest taking approximately fifty minutes and spanning two videos. In most cases, incidents are resolved on camera within a single video episode, though some span multiple films and several incidents are resolved off camera. For a fuller description of the methods used in analysis, see Chapter 4, Section 4.4.2

Incidents were selected for reporting because they represent a cross section of the various kinds of development tasks that characterise encounters with error. They also illustrate different aspects of handling. Finally, they are incidents that are brief enough to be presented in total or near total entirety, allowing the reader to form a sense of how error handling unfolds from start to finish.

Excerpts of dialogue from videos are presented to illustrate aspects of handling. Dialogue that does not pertain to the immediate topic has been removed for brevity and clarity. Exchanges are presented in italics, with the name of the speaker in a strong font. A catalogue of incidents is given in Appendix C.2. Full exchanges are given in Appendix C.3. For fuller transcription conventions, see Appendix A.

6.3.1 Slips of Action

Actions sometimes do not go as planned, or were not intended. They are often simple, routine, and are commonly detected in the act based on perceptions that arise while doing something (Sellen, 1994). Often described in software engineering in terms of backtracking (Bowdidge & Griswold, 1997), they could also be described as slips of action (Norman, 1981). Selecting the wrong item from a drop down menu or improperly referencing a variable are two examples:

***Marcus**: Oops, that's not what I want to do. (Ep. 12, 00:04:45)*

***Joe**: No can't do that cause it's. Oh we can move it outside the [try block]... (Ep. 7, 00:06:51)*

In these cases, the full exchanges for which can be viewed in Appendix 6.3.1, each developer gives a clear indication that something is wrong. What should have been done is evident, and recovery is simple. It is likely that Marcus caught his error in the act. Detection is also commonly made by assessing outcomes, and Joe's statement suggests that he may have responded to effects his actions had on the development environment.

6.3.2 Error Handling Illustrated

The exchange given below illustrates an error in software development that requires handling. In the incident, catalogued as 11-B in Appendix C.2, Marcus is writing a test. He passes a piece of text to as an argument to a method. A red bar appears under the text.

Several features of the exchange are notable. As in the examples given in the previous section, Marcus indicates that the red bar was unexpected ("Oooh."). Joe immediately identifies that the problem is with syntax. The text contains an apostrophe, (e. g. "Do something you don't know") which is causing a parsing error. The action was small, and might have been interpreted as a slip, however, Marcus indicates that he is uncertain about how to proceed. Marcus uses questions to engage Joe in higher-level discussion. Joe signals that he is engaged by questioning in turn. He offers a solution that will allow work to continue. He indicates that the proposed fix is tactical ("for the time being").

00:02:55

***Marcus** (Driver): Oooh.*

***Joe** (Navigator): You've (just) got an apostrophe in there. (Chuckle)*

***Marcus**: What do you want to do about that?*

***Joe**: What? The apostrophe?*

***Marcus**: Yeah.*

> ***Joe**: Umm. Don't, uh, don't use it for the time being. That's a, we'll come back to that later if it becomes a problem. /Okay. / It shouldn't be difficult to remove non characters.*
>
> *00:04:44*

The error is unexpected and the source of the red bar must be identified. Though the interruption is brief, forward progress stops while a solution is identified and implemented. This issue is not exclusively skill-based. Marcus indicates that he doesn't know what to do, but it is likely that he also recognises that the error is technically simple to resolve. Recovery requires decision making that is both ***tactical*** and ***strategic***. The pair need to come to a decision about what to do now, but they also have to consider larger goals for the project.

Joe's admission that the pair can "come back to it" if necessary demonstrates that the issue has a strategic dimension. Joe provides the rationale he is using for accepting the risk. It should not be difficult, he reasons, to handle non-characters. This statement links the issue to one of the global aims that have been given for the project, to allow users of the acceptance test wiki to write tests in natural language. Though this particular problem does not reoccur, the global aim factors into several other incidents that come up over the course of different sessions.

6.3.3 Error-Driven Practice

In keeping with test driven development principles (Ambler, 2012), Marcus and Joe write tests for intended behaviour that initially fail, and are then proven within the implementation. They rely on the environment to "catch" errors ("It needs to go into the right package... If we're wrong, it will tell us."). Similarly, errors thrown within the testing framework are taken to indicate what to do next.. The developers use error-driven tactics to guide work along the way and also to provide placeholders or bookmarks between sessions. Unit tests are left at the close of a programming session that fail, as are accep-

tance tests that fail with stack trace output in the wiki. As Marcus comments in the first minutes of one episode:

> *"That's what I love about ending the last thing I was doing with a failed test cause then I come back and I know exactly what I need to do next."*

There is also evidence that errors are sometimes spurred. An example of this tactic is given in the exchange presented in Appendix C.3.5. In this case, one of the developers might write something in the code that he knows will result in an error. He does this because he has an idea of what needs to be done, but does not know how to achieve it. He is counting on the feedback from the system to circumstantially guide his practice.

Marcus and Joe expect problems to be signalled by system responses (Lewis & Norman, 1987): red bars under method calls or arguments, error messages in the problems pane, or stack traces in the web browser. In spite of this, and as seen in the exchange given in Section 6.3.2, errors frequently come up that catch the developers by surprise. Error handling across the incidents is most often triggered by cues in the environment (Reason, 1990), summarised in Table 6.9, below.

Lewis and Norman described six kinds of system responses that can be designed into software to facilitate error handling during use. As noted in the findings, gags and warnings provide clear indications that something is wrong, either by limiting or cautioning against future action. However, a system might also do nothing, it may self-correct, suggest that the system and user "talk about it", or that the user teach the system what should be done, features that have been explored by researchers investigating new ways to support tool-based development (Ko & Myers, 2008).

In this catalogue, the system responses to which Marcus and Joe respond are often subtle. The display of a screen may not "look right", prompting suspicion that something is wrong. As has been noted in other studies (Lawrance et al., 2013), the developers are also designing and populating messages for system responses. They notice when a

message does not contain information they expect to see, or when it has been improperly interpreted (for a fuller discussion of subtle environmental cues, see Chapter 8, Section 8.1.1). At times, detection is made as a result of interaction between the developers and one of the Watchers. In one case, Marcus notes while explaining designed behaviour to a Watcher that it does not "feel right". In a second, a Watcher points out a weakness in the use of a feature of JAVA.

System Response	Description
Red bar	A red bar appears under a variable or statement.
Problem pane	Most commonly, this pane is used to access errors reported by the unit test framework. The unit test framework places a visual icon at the point in the code where the error is made, however developers tend to indicate that an error has occurred using the textual output in the problems pane of the IDE, and not by making reference to the icon.
Stack trace output	In the wiki that fronts the acceptance testing framework. These appear when the acceptance test requires a piece of functionality that has not yet been written. In general, these errors are used to identify what to do next, but in the case of things that have gone wrong, the error that appears is not what the developers expected to see.
Wiki	Other error messages that come up while executing tests.
"Output captured"	Code can fail in the course of acceptance test that does not result in a stack trace. In these cases, error messages are captured from the failing component and displayed on a separate webpage. The output that has been captured is indicated to the developers with a graphic.
Wiki Error Message	In the web-based acceptance test framework, system responses returned by the wiki.

Table 6.9: Sources of system responses. In this dataset, the following system responses signal to the developers that something has gone wrong. These broadly correspond to the system responses outlined by Lewis and Norman (1987)

Marcus and Joe always indicate that something has gone wrong, usually with verbal comments. In a small number of cases, visual confirmation of the error accompanies verbal comments. The screencast may show a cursor moving along a message or hovering over a red bar to reveal additional information. Often the pair verbally indicate that something has

gone wrong with simple utterances including “Oooh”, or “Oops” or exclamations like “Ugh. That’s interesting.” Questions are asked that refer to the appearance of the system response (Why is that red at the moment?), that personify the code (Why is that complaining?) or more generally question functionality (Why is that not working then?).

6.3.4 Handling in Context

The handling process is managed by how Marcus and Joe assess situations, the information they draw upon, and the mechanisms they try. Attention is often commanded because conditions are unexpected or new, but a situation can turn out to be familiar. Subsequent handling may draw upon knowledge gained through previous experience. The following exchange drawn from Episode 2, and provided in full in Appendix C.3.2, demonstrates both perspectives:

> ***Marcus**: Now this is something to do, I had to solve this recently and I can't remember how I did it.*
>
> ***Joe**: It's an import, you need to import it, don't you? Or it needs to be umm, oh wait, its trying to execute that as a--*
>
> ***Marcus**: --It’s the, the look. There's a, I did this before. It's to do with the way it does the test running stuff. Let's just have a quick look [Driver opens Eclipse] in examples that we were messing about with hums. (Ep. 2, 00:20:43)*

Guessing is a prominent feature of the incidents in this dataset. Guesses are informal and pervasive. They are both right and wrong, often made solely in response to perceived effects. Joe responded to the error message by making three swift guesses about what was wrong. Later in the exchange, he takes a fourth stab about which configuration file will contain the information. All of the guesses Joe makes in this incident turn out to be incorrect. They are an indicator of novelty, and suggest that Joe has encountered a problem that will require conscious problem solving.

Sometimes guessing is used to propose tactics or mechanisms that have an informational basis. In the exchange given above, Marcus takes an informed guess to look at configuration files within the IDE. The handling process he follows demonstrates that knowledge drawn from prior experience is often of limited utility. The environment does not always effectively guide problem solving. In this incident, configuration files are located in different parts of the JAVA project filesystem and they all have the same name. Marcus knows he needs to look in one of the files, but he does not know which file will contain the information. He takes guesses about which files to open and examine. The identification of a solution, as in this case, is often also perceptual. Marcus had to open and look at several files before recognising information in one that he perceived was correct ("That's the one I wanted").

Errors may be encountered together, but they are handled individually. One member of the pair may indicate that a problem is new, while the other may indicate that it is familiar. In this incident, Marcus was able to identify behaviour in the software that the configuration manages, but he didn't explain exactly what the configuration does. Joe's experience was new. His understanding formed by watching how Marcus handled the problem: the actions he took in the development environment and the connection he mades between information stored in the development environment and the wiki.

Likewise, the two do not always notice that something has gone wrong at the same moment, or attribute the same significance to system responses or behaviour in the software. Information is often freely given, but not received: the developer at the desk may not respond to suggestions given about actions to take or warnings about problems. At times, each developer appears to privilege behaviour in the environment over what he is told, only making a detection once he can assess effects. Thus the same error may be caught in the act by one developer, but be detected based on outcomes by the other.

Differences in rates of understanding are not necessarily evidence of differences in reasoning skill or expertise. They serve informational purposes in paired work. Dialogue and commentary are important sources of feedback. Comments can focus a partner's attention, correct an assessment, or trigger an evaluation. The act of explaining a choice triggered detection in one case. Evidence was also given that pairs guide each other on occasion, dictating changes to be made in the code. The steps in these cases are not intended to produce a recovery for the error. Instead, they are given to stabilize the process, restoring immediate behavior so that problem solving can continue.

6.3.5 Modulators

Questioning is prominent in the incidents, found in the context of guessing, but also of other modulators like doubt and blame. Marcus and Joe ask questions when they are not able to make sense of a situation. They question the behaviour of the software ("What is going on?"), the stability of the environment ("What has changed?"), and the location of resources ("Where is it?"). They are doubtful about actions they have taken ("Oh that was the wrong page, wasn't it?") and about where the source of the problem might be ("I wonder if it's like, no?").

Blame is used to deflect responsibility ("That's nothing to do with us."), and to probe for information about a potential source of the error ("What have you done?"). Blame often targets limitations of the environment. In this sense, it functions as an invocation of an external constraint (Guindon et al., 1987), allowing the developers to set boundaries on responsibility. As noted by Eisenstadt (1997), blame is sometimes misdirected, but in this dataset, it appears more often to be a feature of setting boundaries for investigative activities. In these videos, the laptop on which work is being performed is commonly used in this way.

Blame may be associated with knowledge transfer between developers. Throughout the episodes, the pair is working on Marcus' machine, and there is some evidence to suggest that in between episodes, development software is installed and updated by Joe. Given this context, blame may be placed in order to draw out information about the work that was performed, particularly when there is evidence to suggest that it may have a role to play in something that has gone wrong.

Emotive statements like doubt, blame and questioning are sometimes indicators that the developers have entered a "turbulent area" (Amalberti, 2001, p. 119). Within exchanges that are not stable, such statements are recognisable because they are often partially articulated and do not indicate directed reasoning. An example of a severe incident with evidence of blame can be read in Appendix C.3.3. Severe incidents begin like other error handling processes. Marcus and Joe use established tactics and mechanisms, for example by gathering information, or verifying that they are looking at the right page in the web browser. They also manipulate the environment, doing things like flushing caches, stopping and starting a web server or reloading web pages. However in severe handling instances, these techniques do not work.

The severity of issues is linked not to the amount of time an issue takes or the number of tries, but to how "in control" the developers perceive themselves to be. In the most severe incident in the catalogue, Marcus and Joe encountered an incident two thirds of the way through a filmed episode. The issue took over the remainder of the episode, and was not resolved in the whole of the next film, captured on the same day. In total, some 50 minutes of filmed time were spent in problem solving to identify what was wrong.

In this incident, problem solving efforts progressed from very simple examination of files, to discussion about design commitments, and consideration of prior work that used similar principles. At one point, the developers were so flummoxed by the error, and by

incremental unexpected outcomes produced during local problem solving, that they resorted to adding a println() statements to code that would appear in output. Their level of stress and doubt were high, one of them remarking that such tactics were needed to allow them to "prove" that the basic "laws of programming" were intact.

In fact, the very first guess that the developers made about the source of the error brushed up against what went wrong: the framework has a dependency on a jar file that is placed at runtime into the filesystem. The developers were aware of this, and checked to see that it was in place. They failed to notice, however, that they checked for the existence of the file in the ***wrong place***.

6.3.6 Rules-of-Thumb

Some error handling incidents, like those reported in Section 6.3.1 are handled almost instantaneously. Evidence suggests that tasks were simple and the error truly was just a slip of the hand, or a momentary misapplication of attention. However, in other cases, the evidence suggests that the task is not so simple and that the error requires procedural knowledge. There are examples in the catalogue of incidents handled in this way that draw upon well-formed prior knowledge. In some cases, the prior knowledge may match the current situation so closely, it can be applied as a "recipe" or rule (Rasmussen, 1985):

> ***Marcus****: So we have a problem there...that I've noticed happens sometimes. If you actually stop it, now go back to Eclipse and stop it. And then start it again... (Ep. 1, 00:08:18)*

Recovery using the rule is straightforward. Marcus has seen the issue and is able to provide Joe with a mechanism for recovery. The solution is clear, but the circumstances surrounding the issue's earlier occurrence are unknown: Marcus does not indicate how difficult it was to solve, what was tried or how long it took.

To understand how such knowledge forms, it is necessary to compare data from different incidents. The catalogue includes instances of the same error occurring in three different films that were made on different days. The full exchanges for each of the three instances and the prior example can be read in Appendix C.3.4. In each case, Marcus and Joe do not extend an exception class when it is created to satisfy conditions in a test. Here is what handling looks like the first time the error is signaled by a red bar:

Joe:*...why is that complaining? Oh that's because we haven't got the constructors.*

Marcus: *That's right.*

Joe: *Oh, no, that's not, it says it's not a subtype of Exception. Oh [The class giving the error is opened]--*

Marcus: *--'Cause it doesn't extend RuntimeException (Ep. 7, 00:02:57)*

Detection in this case is delayed, spurred some time later when Joe tries to throw the exception. This kind of error could be interpreted as latent and analyzed deductively to determine the cognitive failure that led to its introduction in the code (Ko & Myers, 2005). However, it is also possible to follow problem solving forward. Joe makes a guess about the source of the problem, signaling a shift from detection to identification and the pair undertake a brief cycle of local problem solving (Reason, 1990) to identify what is wrong.

The value of prospective analysis is made clearer by examining the subsequent occurrences. The second time a detection is made, the issue is familiar. Circumstances are slightly different; this time Marcus is adding a constructor to the exception class when a red bar appears. Joe is able to swiftly identify the source of the problem, and he takes responsibility for the error. He indicates that it might have been avoided:

Joe: *Oh, that's 'cause it doesn't extend runtime. I was lazy and I didn't (inaudible).*

Marcus: *But do you know what? Actually, ...I think now is the right time to actually put that in there. (Ep. 11, 00:16:53)*

Joe explains that the error was one of omission and that the class had not been created with strategic oversight (Reason, 1990). However, Marcus counters that the omission is acceptable, because it upholds a preferred practice. This instance represents an example of the development of know-how or the formation of a rule-of-thumb. Rules in this sense are cultural (Rasmussen, 1985), a point that is emphasized in these exchanges. The pair may be following principles of test-driven development and object-oriented programming, but reserve the right to determine how classes are managed in relation to one another, even if this results in an error that reoccurs.

Joe's handling the third time enforces the practice and demonstrates the prior knowledge he has gained. There is no additional dialogue about how the error should be handled. It is still unexpected, but familiar, and handling has become routine. It is an error that can be caught more or less in the act and one that can be quickly recovered from using a rule.

> ***Joe**: Ahh [a red bar appears in the IDE]. So we didn't include the, when we created it we haven't made it extend exception. So now to make it... runtime exception. And we need a constructor with a message... (Ep. 18, 00:15:26)*

Taken together, the occurrences reveal three things. First, they demonstrate that ***responses to error are modified over time***. They also demonstrate how ***preferred practices*** are formed. Rasmussen described this as the formation of a rule or rule-of-thumb. He indicated that these rules are cultural (Rasmussen, 1985), a point that is emphasised here. Finally, these occurrences also demonstrate that ***error occurrences are not always avoided*** (Brodbeck, et al., 1993). The fact that the incidents repeat and are so similar suggest that the developers see little of value in this kind of error; it provides them no impetus to change behaviour to prevent subsequent occurrence. Instead, they place greater value in consistently following the practice they have adopted.

6.4 Discussion

This section builds upon the rubric given at the start of Section 6.3. It gives a fuller characterisation of error handling in software development at the desk, drawing evidence from the points discussed in the findings, and from other instances in the broader catalogue. For reference, it may be helpful to consult Appendix C.3, which gives full exchanges for incidents.

Error handling begins with detection, with knowing that something is wrong. Errors are detected in terms of perceived discrepancies between intentions (What I mean to do), expectations (What I expect will happen) and outcomes (What actually happened) (Reason, 1990). Marcus and Joe use and rely upon system responses to organise practice and detection is thus usually based on an assessment of outcomes (Sellen, 1994). There is evidence in the catalogue of self-detection within acts, and of errors that are detected by the other member of the pair or by a Watcher.

Marcus and Joe invariably indicate that they recognise that something has gone wrong, responding to system responses with verbal comments. Sometimes the significance of an outcome may relate to visual or communicative elements in the interface that are significant to the developers, but which might not be evident to an observer. Corroboration given by the developers indicates that development activity has shifted from problem solving undertaken to do something, to looking over what was done previously (Allwood, 1984). In terms of error handling, corroboration indicates that developers are aware that a problem exists and that the handling process has shifted from detection to identification.

Once they realise an error has occurred, Marcus and Joe must identify what was done wrong, determine what should have been done, and take steps to remove the effects of the error. As the examples show, error handling is often simple and compressed. However, sometimes recovery requires several rounds of local problem solving (Reason, 1990).

Guided by system responses (Lewis &Norman, 1986), information gathering (Eisenstadt, 1997) is interspersed by manipulations of the environment. The process is action-based (Norman, 1981), the developers set local, immediate goals, and identify actions that can be undertaken, observed and assessed (Reason, 1990; see also Chapter 2, Section 2.2.4 for a summary of Reason's GEM framework). Evidence is given that Marcus and Joe search for commonalities between prior experience and the current situation (Rasmussen, 1985), but also that they express uncertainty about how to proceed and communicate that they do not understand what is wrong. An example demonstrating local problem solving can be read in Appendix C.3.5.

Within software engineering, a similar process has been described as being "bottom up". In the stories Eisenstadt gathered, the developers may have had a rough idea of what they were looking for, but they were not systematically testing hypotheses. Instead, they were ***gathering data*** (1993). More recently, this has been described as ***information seeking*** (Ko et al., 2007) or ***scent-following*** during bug-fixing, with sources of information reported as being source code, run-time information, and the internet, among others (Lawrance et al., 2013). These authors, unlike Eisenstadt, differentiated the process of finding what was wrong from identifying what should have been done, a process they describe as ***fix-the-fault*** and which they note took longer than any other bug fixing activity.

Error handling is often required when conditions and situations are novel (Norman & Shallice, 1986), when something comes up that has not been seen or done before. This is true even in the context of error-directed practice. Handling is guided by the techniques Marcus and Joe use to develop their frame of reference toward the problem (Rizzo, Ferrante, & Bagnara, 1995). Information is gathered by looking at files, by examining system responses, and by reflecting on prior work. Using these sources, the developers

confirm understanding of how related components work, identify areas of code to alter and generate ideas for possible sources of problems.

Information gathering is interspersed by manipulations of the environment. The pair change algorithms, class declarations, tinker with syntax, stop and start tools, alter configurations, flush caches. These mechanisms generate changes in the environment that must in turn be assessed for their effects. Handling is punctuated by things that do not work: making changes to code that do not remove the problem, looking in a file for an error and not finding it, stopping and starting a server to no effect.

Mechanisms that may fix the problem are proposed by developers at different points in the handling process. Sometimes the correct mechanism is suggested in response to the detection, but may not immediately be employed. At other times, the same fix is proposed more than once, punctuating other investigative activities that turn out not to work. Though the successful removal of a system response is often noted, the fix itself often is not remarked upon at recovery.

The process is also modulated by emotion. Marcus and Joe ask questions, express doubt, they are suspicious and lay blame, they take guesses and make declarations. However, the pair are rarely critical of each other. It is much more common for them to be critical of the environment in which they are working, or to use blame to elicit information about changes made to the environment by tools or by the other member of the pair.

Error handling can be prolonged. A single sequence of activity may represent the entire process, however some occurrences thread through the completion of other tasks. These issues invariably relate to “higher-order” concerns such as how to define conceptual boundaries for classes or how an object in a model should be expressed using features of a language. Incremental progress is made through verbal consensus that satisfies the

developers and permits the issue to be set aside. In all cases, a subsequent instance occurs in which changes are made to the software.

Recovery is not always permanent or complete. Evidence of compromise in solution selection is given, of deferring solutions by addressing symptoms, and of backing out significant changes. In the case of particularly severe issues, problem solving may be aborted. In these cases, the interruption to development is significant; the handling process may take longer. Filming on the day may cease, with the developers providing a short explanation in the following episode about what was found to be wrong and how it was fixed.

The aim of error handling is to return the software to a running state so that work can progress. Marcus and Joe do not always indicate that they understand what was wrong or why a particular mechanism yields a recovery. Gaps in understanding are also revealed in instances in which a recovery mechanism is drawn from prior experience. This was demonstrated in the findings by juxtaposing how prior experience for one developer accompanies a novel experience for the other. Joe did not need to understand why the mechanisms given to him by Marcus fixed the problem; he only needed to employ them. Prior experience is useful if can be used to direct similar processes that occur later.

The suggestion is given that gaps in understanding are acceptable and that fragments of knowledge are sufficient. Beyond acknowledging that something is “strange” or “weird”, the developers do not always exhibit curiosity to learn more. In most cases error handling is successful. The pair are able to continue working, suggesting that an identification has been made, and a change can be made in the code, in a configuration, or within a tool that will remove effects.

6.4.1 Limitations

This study depended upon secondary sources that were gleaned for data. As with other studies that leverage paired work to gather naturalistic data (Bowdridge & Griswold, 1997), it was not always possible to establish motivation and meaning for the actions depicted in the films. Explanations and justifications for activity were provided when the interaction between the developers demanded it, not in order to meet research protocol requirements.

The limitations observed in the videos used in Chapter 5 were augmented in this study by elements of the production. The camera depicts a screencast of the machine on which the developers are working. As a result, it was difficult at times to discern who is driving or how work performed in different episodes relates. There were gaps between tapings, during which conversation and programming occur that are only obliquely referred to on film.

Intermittent, various technical difficulties made comprehensive analysis difficult beyond episode 20 (for a fuller description of the corpus and processing, see appendices C.2 and C.4). Sampling of the later episodes suggests that the quality of the discussion changes, with fewer brief incidents and distractions from third party software and hardware. Changes in quality of discussion might have a bearing on how errors would be characterised in analysis.

6.5 Conclusion

This chapter reported a qualitative study undertaken to examine how developers deal with error in day-to-day work. It examined the activities of two developers performing tasks over the course of a month on an open-source programming project.

Error handling ***suffuses development practice***. It is required for all kinds of activities at the desk. Errors occur when developers specify behaviours in tests, while they implement

classes, in periods when they first implement functionality and when they refactor. They occur in relation to software that is being used and written. The aim of error handling at the desk is to move forward in development, and is predominantly cued and directed by ***system responses*** (Lewis & Norman, 1986).

Though in some cases developers indicate awareness of what they are trying to achieve for a project, what comes through most strongly during error handling are efforts to ***understand minute, material details of the environment*** in which they are working. When the developers in this study encounter an unexpected system response, they behave in ways that are consistent with other findings of problem solving at the desk (Kristoffersen, 2006). They solve their problem by assessing what is before them. They try to understand what they are seeing ***at this point***, and only gradually, as necessary, expand their investigations to higher-order concerns about features of programming languages, architectural concerns or design.

Instances were identified and interpreted in relation to stages of detection, identification and recovery. Data in the catalog reflect the broad characteristics of error handling as conceived in research from psychology and illuminate how developers consider and manage local and global aims during problem solving. Error handling is influenced by prior experience, and modulators that include guessing, doubt and blame. The severity of issues, as in Chapter 5, is revealed by evidence of turbulence: problem solving that includes many of the same factors as normal handling but which gets out of hand. Rules of practice develop between developers over time. However, even in collaborative work, errors are experienced individually.

7. After the Fact

When errors come up at the desk, developers must assess what is before them to ascertain what is wrong and how to remove the effects of what they are seeing. Very often, the errors that are encountered relate to the behaviour of tools and software that is being used, though they may also relate to lingering conceptual errors. Errors arise in the context of actions taken to implement behaviour using features of programming languages and those taken within libraries and frameworks that are being used. While global project aims figure into handling, error detection and recovery are more often concerned with managing immediate, material conditions that arise in the working environment.

Software development is managed through process but is continuous, "embedded in everyday work practice" (Dittrich, 2009, p. 394). The experience of individual developers is likewise continuous. Software takes time to write, developers often work on multiple projects serially and concurrently. They bring to each day understanding (Winograd & Flores, 1987) formed out of prior experiences that can stretch back in time for many years.

A developer's state of mind at any moment is inherently ephemeral (Eisenstadt, 1997), and the errors they encounter are likewise experienced, they are active and fleeting (Reason, 1990). They leave few material traces (Scott, 1990) within code, descriptions or project records. The meaning associated with them is personal, shaped by passing time and the social and organisational boundaries that form the space in which workers perform (Rasmussen, Pejtersen & Schmidt, 1990).

The following pages of this chapter report a study that examines how developers recount problems in recent work. The aim is to explore individual experiences with error handling within the broader timeframes and situations that characterise software development in professional contexts. The chapter begins with a brief review of related work. The

scene is set for two sites at which interviews were collected. An analysis is given of six accounts, followed by a discussion of the nature of errors in organisational work.

7.1 Related Work

Software engineering research generally reports the experience of developing code sparingly. A small number of sources provide descriptions of ***what it is like*** to write code. Though written for different purposes, these sources include reflection about strategies, successes and failures. Turkle's *The Second Self* includes profiles of hackers and of maintainers of early personal computers, who are "intensely" involved with computers (Turkle, 2005). Oral accounts have been taken to provide a glimpse into professional life (Lammers, 1986). Other first-hand accounts describe the experience of language development (Krasner, 1983) or of developing a piece of software over time (Knuth, 1989).

There is growing awareness within software engineering of the power of rich accounts to illuminate aspects of practice (Sim & Alspaugh, 2011). They have been used, for example, to gather stories about "hairiest" bug fixes (Eisenstadt, 1993, 1997) and to learn how developers "really" use documentation (Lutters & Seaman, 2007). Accounts are useful because they are "phenomenological" (Eisenstadt, 1993, 1997), they can be used to develop understanding of how developers think, and what they experience.

Reflecting on the experience of writing the first version of TeX, Knuth described how he encountered "loose ends" in the design, an outcome that ran counter to his understanding heading into the process. Though he had imagined that the specification was "quite complete", the process of writing the code involved ***twists and turns***, requiring that "policy decisions" be made (Knuth, 1989, pp. 612-613). He concluded from this that designers of "new systems" must participate in implementation.

Following Knuth, Eisenstadt collected anecdotal accounts via email of professional developers' "thorniest" experiences with bugs (1993, 1997). After performing an inductive

analysis of the anecdotes, he pursued a more detailed analysis to examine why the problem was perceived to be difficult, how the error was found, and what the developer perceived the root cause to be. Eisenstadt found that most bugs were found either through "bottom up" data gathering, or by "thinking about" the code. Other categories included those that were found with the help of fresh eyes or through controlled experiments.

Accounts of debugging practice have also been gathered using interviewing techniques. One notable study that collected accounts from developers was conducted at Bell Labs to produce a training course to promote expert debugging skill (Freeman, Riedl, Weitzenfeld, Klein, & Musa, 1991; Riedl, Weitzenfeld, Freeman, Klein, & Musa, 1991; Weitzenfeld, Riedl, Freeman, Klein, & Musa, 1991). Like the report given in this chapter, data were drawn from critical decision method interviews. Interviews were taken with experts who were selected after consultation with managers. Findings were corroborated and enlarged through a focus group and surveys distributed to developers throughout the company.

The study found that expert debuggers think before taking action, wait longer to employ debugging tools, and seek information about what to try next rather than jumping into "poorly directed" but hopeful activities. Less experienced developers, by contrast, were perceived to thrash around, to follow an ineffective process of going over and over a problem. ***Thrashing***, not to be confused with the term used to describe memory management on hardware, was described as a negative novice behaviour of no perceived value. Novices were reported to fail to recognise when they thrash, and to be unable to break out of it. Experts, on the other hand, might thrash, but are able to attend more quickly to emotional cues that they are doing it, and to seek help sooner from colleagues with greater expertise.

Experts and novices were found to employ "close reading" to establish what code does, but they responded to information in different ways. Novices were less critical of what

they read, while experts treated the comments as evidence of the number of hands present in a piece of software, and to signal the conditions under which developers were working when code was written. Experts read code as a last resort, preferring instead to seek help first from other developers with detailed knowledge of the software.

The study collected detailed information about technical aspects of bug fixing, however the aims of the study were to develop a training course. Because of this, feedback given by participants led the analysts to focus their efforts on explicating the social aspects of debugging. Likewise, the researchers did not examine expertise in the context of other kinds of development activity. Findings were reported based on the views of a small number of developers from a single organisation. Data were collected primarily from experts, which may have affected findings related to differences between novice and expert behaviour.

7.2 Setting the Scene

This section sets the scene for the analyses given in the findings. It presents an overview of how work is organised at the two sites that were visited. In this section, quotes from informants are given anonymously.

This section does not include a full account of the methods that were used for collection and analysis. For this information, see Chapter 4, Section 4.3 and Section 4.4. It may also be helpful to consult Appendices A and D.

7.2.1 Digital Humanities (Site B)

The developers at Site B work in an established digital humanities centre at a university in the United Kingdom. Digital humanities departments use new media and technology to support humanities-based research, teaching, and to promote "intellectual engagement and experimentation" (Zorich, 2008, p. 4).

Seven people were interviewed: six men, and one woman. Each developer was asked to recount an incident from recent work, in audio-recorded sessions that lasted from between forty-five and seventy-five minutes. Interviews were conducted at the desk; five of the people interviewed were located in a large, open plan office. For an example of one of the offices, see Figure 7.1. Desks were clustered together and informants were within hearing and sight of one another. Not everyone in the office was interviewed; all of the people located in the office were aware that interviews were being conducted. The sixth developer was located in a different open plan office, and the final interview was held in the informant's private office.

Developers old and new to the organisation were interviewed, with one having less than a year at the organisation, and one more than ten years (see also Table 7.1, below). Two informants had computing degrees, one had a computing postgraduate degree, one had a computing applications postgraduate degree, and one had a postgraduate computing diploma. Three had industry computing experience in the web media, financial, education and GIS sectors. Two had post-graduate or research degrees in the social sciences and humanities.

There were also humanities computing specialists, with one informant having at least two twenty years of experience in digital humanities work, and a second having a decade and a half. These informants had worked in multiple organisations on digital humanities projects. For the other informants, the current position was their first in a digital humanities centre.

The choice to conduct these interviews *in situ* was deliberate. It was felt that conducting them in the developer's own environment would allow for better access to physical and digital artefacts mentioned in conversation. Given the topical focus, it was also hoped that holding discussions in the open would signal to informants that the purpose was not to

establish or assign blame. Informants gave no indication that the choice of venue made them uncomfortable, though in several cases individuals displayed discretion in referring to colleagues who were located in the same office, either by lowering their voices or by referring to them simply as “my colleague”.

Site	Name	Gender and Age	Experience
Digital Humanities (Site B)	Joachim	Male, thirties, 5.5 years	Computing, Educational Software, 10yrs.
	Evan	Male, thirties, <1 year	Computing postgraduate, GIS, 5yrs.
	Valentin	Male, thirties, 3 years	Computing postgraduate, Web Media, Financial Industries, 11 yrs.
	James	Male, sixties, +10 years	Humanities Computing, 20 yrs. +
	Marisa	Female, twenties, 2.5 years	Humanities + Postgraduate computing diploma, 3.5yrs.
	Richard	Male, Forties, 1.5 years	Humanities Computing, 15 years

Table 7.1: Informant Demographics, Site B. Detailed accounts are given in Section 7.3 for Joachim, Evan and Valentin, the informants highlighted in grey. The accounts from the other three informants were used to characterise how work is organised at the site.

Information on computer screens, paper diagrams and a poster on the wall were used to initiate discussion in three cases. In addition, informants shared source code with the interviewer, explained the output of stack traces and demonstrated debugging tools, prototypes and software under development. Several developers appeared to remember with their fingers, verbally recounting details while at the same time accessing files and

websites and conducting internet searches similar to those they had used while solving problems.

Figure 7.1: An open plan office in the Digital Humanities Department (Site B).

7.2.1.1 How Developers Work

Developers in this centre tend to work alone, even when assigned to tasks for the same project. A single person may be assigned to work on all deliverables, or different people may be assigned to different areas of the software. It is common for developers to perform tasks periodically for the same project over time. Developers know the others who are working on their projects, and report that they attend meetings at which other developers are present, however each works in reference to the overarching project team which is led by domain specialists.

Informants identified particular technical expertise such as in application or interface development or in data modelling. Despite this, several recounted the need to learn new skills to meet requirements for projects that emerged after the project had begun. For example, one application programmer described learning and implementing client-side

technologies, while another developer who was proficient in XML data modelling described a need to learn relational data modelling.

Developers also take the initiative for prioritising and organising their work. This can involve adopting new working practices, as in the case of one who described introducing a new working style on his project as "agile-like," with rapid iterations and frequent meetings with project partners. Another explained that his responsibilities at the organisation are growing, and how he extended a recent task he had been given on his own initiative, more or less as "the accepted order of things".

Though developers work independently, there is the sense given that they possess historical and cultural knowledge. References were made to technologies used on projects in the past by different developers., and *ad-hoc* technical teams are formed to solve particular problems. One informant described "finding" his way on a project with the help of an "amazing" colleague who offered technical advice and guidance about how to manage relationships with partners. Another felt the luck he had in finding a solution was due to the technical expertise of a colleague who had not been formally assigned to the project. A third described looking to a trusted colleague for help before relying on internet fora and other technical documentation.

7.2.1.2 Projects

Projects at the centre take a similar form: tools are created for use by humanities scholars who need to manage and create data related to physical, often historical materials. These data are in turn presented to the public using other pieces of software that are also developed by the centre. Public facing outputs take the form of web editions of texts and web reference tools. In some cases, monographs are also published.

Developers assigned to project teams produce software for both of these user groups. Scholars that are performing research within projects are prepared to work with tools that

require complicated installation procedures or which have a less than finished feel. Their priority is to have a piece of software finished enough so that they can advance their research. By contrast, readers of public-facing web editions and reference tools, themselves also typically domain specialists, have an expectation that the tools they use will be finished to a very high standard.

Developing research tools for the public, and doing so in new and innovative ways, is a central priority for the centre, but the requirements for these tools emerge slowly, sometimes over a period of years as the specialists work with original materials and interpret their meaning and significance.

Joachim, Valentin and Evan (detailed accounts for whom are given in the Section 7.3 Findings, below)described incidents related to work performed for two projects. Valentin described a project for which he was the sole application developer, tasked with creating both an editorial tool and a web edition for displaying a critical edition of texts (summarised under Legal Texts in table 7.2). Joachim, and Evan described performing different tasks for a single project to support detailed annotation and display of medieval handwriting (summarised under Medieval Handwriting in Table 7.2 below).

Person	Project	Description
Valentin	Legal Texts	Research Project, Editorial tool, and web-based critical edition of Legal Texts
Joachim, Evan	Medieval Handwriting	Research Project, Editorial tools, Web-based annotation tool.

Table 7.2: Projects, Site B.

The data produced and managed to support humanities research are different from commercial data: they are less structured, orientated around natural language and approximate. One developer characterised them in this way:

> *"So a good example are dates. If you say the date of this manuscript is around 1113 well it could be this date or it could be that date. Or even worse somebody is saying it is that date, somebody is saying it is that date, somebody is saying it is that date. In the commercial world it is just a single precise date to the millisecond. Here you want many dates by different people and you want all the opinions shown on your website and preserved. So the interpretation is very important."*

Every informant at Site B reported a working pattern of "fits and starts", the need to pick up a task and set it down as required to meet the demands of multiple projects. The issues described by informants included relevant details that were at times temporally distant from one another. Projects have relatively long time frames, some lasting three or more years. This means that developers have more freedom to take time with issues, but they are not free from time-based constraints. One developer indicated that he felt pressures on his time, the need to "get something working" for this project, while still meeting the demands of other projects.

7.2.2 Course Planning (Site D)

Developers at Site D are employed in the information and communication technology (ICT) department at a public university in the United Kingdom. The department is developing a new set of web-based software tools for student administration and curriculum management. The team develops a subset of the services related to course management. At the time the interviews were taken, the department had recently re-organised, adopting Agile development practices centred around Scrum. The products under development were not yet in production, though some were nearing completion. In the month during which interviews were taken, a test release of a small component was made to the wider university.

Access was granted by an internal manager responsible for multiple development teams. This manager circulated an email to developers that introduced the study and invited

participation. The email communicated that developers who took part would be allowed to "cost" an hour of working time to the interview. Four application programmers responded to the invitation and were interviewed in sessions that lasted between forty-eight and fifty-eight minutes. All of the developers belong to a single team within the department.

As in the visit to Site B, all informants were asked to recount a story about a recent problem. Each informants met at their desk, but interviews were conducted in other places. Three interviews were conducted in public areas within the university. One was conducted in a small meeting room adjacent to the desks.

Each developer created a diagram in the course of discussion, one voluntarily and two upon request. These diagrams depicted aspects of screens and component models. A rough timeline was drafted during the interview that captured points related to time and decision making. Both kinds of artefacts guided conversation; the developers used the diagrams to explain how screens looked and behaved, and to relate aspects of how underlying features of software architecture related to the behaviour of information on screens. They also used the diagrams to explain how elements of the software related to one another.

Upon completion of the interviews, access was negotiated with the team leader to observe the developers in practice. Aims for the observation were to gain a sense for circumstances that led developers to come together. Clear decision points and sequences were not as easy to identify through questioning at this site; informants made extensive use of the term "we" to describe actions taken or decisions that were made.

In addition, though each developer was asked to recount a piece of personal work they had found challenging, two developers interviewed a week apart elected to discuss what appeared to be the same issue. Both indicated that they had solved the issue, both indicated that they had used similar resources to support the resolution. The second person interviewed explained that some of the work had been performed with another team member,

and that he had taken responsibility for the issue when the colleague left for holiday. Observation was made to clarify how team members share responsibility.

Site	Name	Gender, Age Time (in team)	Experience
Course Planning (Site D)	Robert	Male, thirties, 3 months	Computer Science/ Software Engineering, E-commerce, Airline 12 yrs.
	Dereck	Male, thirties, 6 months	Computing & Accounting, Media 6yrs.
	Thomas	Male, forties, 10 months	Degree Unknown, Commercial development 20yrs.

Table 7.3: Informant demographics, Site D. Detailed accounts for Robert, Dereck and Thomas are given in Section 7.3 below.

As shown in Table 7.3, developers have a range of professional experience. Dereck has around six years of experience at work after having taken a degree in information systems analysis. He reported having worked for two small companies prior to joining the department. Robert has a degree in computer science, and has been working professionally for twelve years. For five of those years, he has worked as an independent consultant. He is a certified scrum master, and one of his responsibilities on the team is to help increase knowledge within the team, to "make sure that things are kept moving or progressed a bit quicker than they have been". Thomas reported having worked professionally for twenty years at "software houses" and in other companies. All three have worked on the current team and within the university for less than a year.

7.2.2.1 How the Team Works

The developers sit together in a compact space; the floor on which they work is filled with several similar "pods" of desks. This team uses a storyboard to track tasks for individual

sprints, and also a physical board to manage bug fixing tasks, two techniques that have been introduced by a member of the team who is also a scrum master.

The "bug board" has magnets with photos of developers' faces. The faces are queued on the board, and the first person in the queue is the next to take on a new bug. This physical system was developed in part to work around the task management software in use by the organisation. One informant explained that the system groups tasks of different kinds together and it is not always easy to differentiate tasks related to maintenance or feature requests from bug reports. In addition, application developers have not been granted access to all parts of the system that contain information about bugs.

The team was recently formed and members give the impression that they are still getting to know one another and the department. References to past development decisions are criticised, but there are indications given that knowledge of what actually happened is vague. Observation was made of the team members informally discussing preferred practices for committing software to the version control system. Commits must happen frequently; the conversation indicated that the team members were not yet familiar with each other's preferred habits.

The sense was also given that the team are still forming their practices; one informant described that the team had recently set a plan to use the whiteboard for discussion, but that the idea had not been regularly taken up.

> *"(T) he idea was in the sprint plan we'd come up with the tasks and then when we got back to the desks as you'd picked a task up then we'd head round the whiteboard but at the moment that hasn't happened".*

Another reported having used a standard practice to send an email to the team to give background to the problem he described. However, when he reviewed his records, he realised that he had not actually followed the practice in this case. When asked for detail on this point, he noted that he tended to be brought into issues that required problem

solving, and that his personal practice was to share what he had learned verbally, through email, or within a documentation tool like a wiki

The team is newly formed, but is situated within a mature department. This is perceived to impact how decisions are made and policies are set. One informant described data modelling practices that differed from those he had been taught, but which reflected how things had been done in the past in the department using older generation technologies. This is the largest, most mature company one informant has ever worked in and he has noticed differences in how decisions are made that he links to the stability of the workforce:

> *"… it takes a very long time for a decision to get made… [Given the nature of] this company people just don't leave…I mean it is brilliant because it means in terms of industrial knowledge it is great, you know the people there they know things, not because they've read things, or because they've been taught things, but because they were there when it happened. So background knowledge in this place is brilliant… there will be somebody within these walls that will tell you everything that's happened in the last twenty years."*

7.2.2.2 Sprints and Tasks

The course planning team is organised using principles of scrum. Work is conducted in two-week sprints, during which a set of tasks to be addressed are agreed to and undertaken. The sprint begins with a task setting meeting and finishes with a review meeting, during which information is shared about particular problems that came up. This meeting is also used to reflect on work practice.

Incidents were described in terms of tasks that had been set for work sprints, summarised in Table 7.4. The relation of the tasks to one another was not made clear in the interviews, nor was any clear sense given of projects to which the tasks belonged. That is,

the incidents may have occurred during the same sprint or in different sprints; they may all have been undertaken as a part of a single or multiple projects.

Two of the tasks involved developing web-based user interfaces: Robert described a task to perform validations on a web form; Thomas described building screens to render form elements in specific ways depending on user actions. Dereck described a task related to web services maintained by the team that provide data to other teams in the department.

Tasks	Description	Person
Client-side validation	Triggering client-side validation in dynamically loaded pages.	Robert
Rendering Forms	Rendering form elements based on actions taken in different screens.	Thomas
From Maintenance to Live Data	Altering reference to a database so that it draws on live rather than testing data.	Dereck

Table 7.4: Tasks, Site D.

Working time is costly and is closely monitored. Agile practices are relatively new in the department, but developers have adapted their thinking about time in terms of scrum practice, referring to past events in terms of the number of "sprints ago" rather than in weeks. Tasks cannot be undertaken unless they have been defined as belonging to the sprint. This can affect decision making during development by constraining the options available

> *"[N]ot for that sprint, because we'd only really tasked a story and estimated doing that one, and we were told not to do this one, could not do that".*

Nevertheless, the team knows how to manipulate time within sprints in order to fit in work that will further development rather than product aims, a point that is exemplified in the consequences of Dereck's slip, reported in Section 7.3.6 below.

7.2.3 Points in Common

The problems reported at the two sites shared technical commonalities. Both sites were working with similar technologies and were building web-based software. At Site B, developers were working on Linux systems with open source tools and standards. To develop websites, the team used a popular open source web application framework and to manage software, an open source versioning repository. At Site D, developers were working on Windows systems and performing development using Microsoft's web framework. The department was also using Microsoft task management and release management software. Both sites relied on open-source JavaScript libraries to manage aspects of client-side behaviour.

7.2.4 Exclusions

Eleven interviews were collected, however, five have been excluded from detailed reporting. One interview from each site was not transcribed. At Site B, the seventh interview did not result in the identification of a clear incident, a view that was corroborated by the informant. This informant also indicated reluctance to be included in reports. At Site D, one interview was conducted in a public area with significant background noise. As a result, it was not possible to accurately transcribe the audio recording.

Three additional accounts from Site B have been used solely to inform contextual understanding reported in Section 7.2.1. Each account suggested the presence of local active, error handling processes. However, the detail provided during the interviews was not sufficient to permit the "active" parts to be discerned. One informant described a sudden, visionary breakthrough in thinking about how to re-architect a piece of software.

Unfortunately, the blocks that preceded the breakthrough were not specifically described. Another described his problem as “the worst thing ever”. His account emphasised the research and design process he followed to meet complicated, ambitious requirements for a user interface, but it was not possible to discern just what he had perceived the “worst thing” to have been.

7.3 Findings

In this section, accounts are given for six of the developers who were interviewed. To indicate that the occurrences belong to the more general category of software development experience, the accounts are presented together, without subdivisions marking the site at which the corresponding interview was collected. Stories are given in narrative form, and are organised chronologically. Subsections are used to draw out particular features of accounts relevant to error handling or circumstance.

The views of individual developers are presented using pseudonyms that were introduced in Sections 7.2.1 and 7.2.2 above.

7.3.1 Settling

Joachim described fixing a recently reported bug. His account was collected in the midst of ongoing work for a project. He seemed to have difficulty in establishing a sequence of linked events. A timeline did emerge, but it was established during analysis. Joachim meets at least once a week with researchers from the project team to discuss issues and requirements for new features. The aim is to get the tool working well enough so that editors can begin using it to analyse texts. The tool is being developed using an open source JavaScript library designed to support mapping applications.

In a recent meeting, Joachim’s editors asked to have keyboard shortcuts mapped to a toolbar of functions that are used to annotate images. The way keypress events are handled in the mapping library would allow editors to delete annotations with a single click.

Joachim was concerned that this would be too error prone for editors. To implement the feature, he "looked around" on lists on the internet for an alternative and picked one he saw discussed that seemed to be the "best one". He implemented the feature in around an hour.

7.3.1.1 When they told me we had this problem I thought what could it be?

A couple of weeks later, a problem with the keyboard shortcuts was mentioned in a meeting. The researchers said it only happened in a single browser. Later they sent him an email with a list of issues and feature requests:

> 2) The icon toolbar disappears when using keyboard shortcuts. e. g. Ctrl-W or Ctrl-R in Chrome. Or Ctrl-C in Firefox.

Joachim began to debug by trying things out in the browser and stepping through the code in a browser-based debugger. Replicating the behaviour described in the report was not straightforward. The shortcuts given in the report were key combinations that had been mapped to icons in the tool bar. When Joachim tried these, “nothing weird” was happening. By “accident”, he decided to try shortcuts that had not been mapped to icons in the toolbar and he realised that the error occurred when the user typed a key combination that did not exist. He could see in the debugger that a variable populated with a method call contained a value he did not expect. At a certain point, he realised that it was only when an unmapped shortcut was keyed two times that the toolbar disappeared.

The fix took less than an hour. It required making changes to a single function. Keyboard shortcuts were being managed using a switch statement, but Joachim hadn't added a default case to manage unmapped key combinations. He also wasn't performing sufficient checks on the state of the objects in the toolbar before activating or deactivating operations.

Joachim names the source of the latter problem as one of understanding. He "wasn't sure what the right behaviour was" when he was first writing the function. He believed that a particular function call in the library would return nothing if the user had not selected a tool for use. This condition was met when an unmapped key combination was entered. In this case, however, the library actually returned the parent object, the toolbar, which his code deactivated.

7.3.1.2 I'm still not very happy with it

Now Joachim has produced a solution that is meeting requirements. Echoing comments made by Bill in Chapter 5, Joachim is not satisfied, explaining that he is “still not very happy with it yet,” and that he is not sure how well the solution is working. In the course of our conversation, several concerns were mentioned related to his satisfaction with the recovery. All of them could be classed as problems of understanding.

He brought to the task a degree of expectation that he would have trouble handling keypress events, linked to requirements given by the researchers. As a user, he has noticed that keyboard shortcuts are not common in web applications and the ones that do use them, like Google Mail, tend to use single key shortcuts, not combinations of keys. By contrast, his researchers have specifically requested that shortcuts be key combinations, so that it more closely replicates behaviour they have observed in desktop applications they use. In addition, some of the shortcuts that have been requested are the same as shortcuts that have been mapped within the browsers. He comments that this overlap can make it "a bit of a disaster" to manage behaviour if the application doesn't have focus when the keypress event occurs.

Though it didn't figure directly into fixing the bug, Joachim is also suspicious about the method within JavaScript he is using to catch keypress events. He mentions several times a lack of confidence in the way he did this, wondering "maybe I'm not doing it the right

way". He based his selection on an assessment of internet sources, but notes that the decision was softly assessed. His strategy was to determine what "seemed to be the best one", by identifying the one that "most people seemed to be using".

Joachim is also cautious in his commitment to the mapping library, describing it as not the right way, but “a better way” to manage annotations than those that have been used on other projects in the department. The library was selected at the start of his involvement in the project, some four months earlier. In that time, he has become familiar with the documentation for the library, which he uses regularly. He feels pretty comfortable working with it, but notes that this is the first project on which he has used the library to do "proper work". On previous projects he had used it only to display images and in those cases, the code had been written by someone else and given to him to incorporate.

7.3.1.3 I thought there ought to be a way to reuse this code

Joachim’s story began as an account of a bug fix, but in relating his dissatisfaction with the outcomes, it became clear that the handling was one small knot in a larger thread of practice.

He is developing a class built around the mapping library to more generally support annotation. The error handling process he described provided feedback to him about requirements for annotation in his domain, web-based interaction models, strengths and weaknesses in the mapping library, and more general information about the languages used to support this "Web 2.0".

Joachim exchanged correspondence with the researcher after the interview. In the following months, he finished the class, and released it to the public under an open source license. As it turns out, he continued to use the mapping library, but reported that he found a better way to manage keypress events.

7.3.2 Tolerating

Valentin described an issue that surfaced as a bug several times over the course of nearly two years in tools used by the developer and in different areas of the software being developed, as depicted in figure 7.2. The issue was related to the use of Unicode, which presented particular complexities in this domain.

He considers himself to be well-versed in using Unicode, however, this is a problem he has never encountered before. His prior experience with Unicode related to databases, or conversion and rendering of texts written in modern languages. He is aware of how to use Unicode to render characters from different alphabets, but has never had to consider whether or not a font would be available that could render the necessary characters.

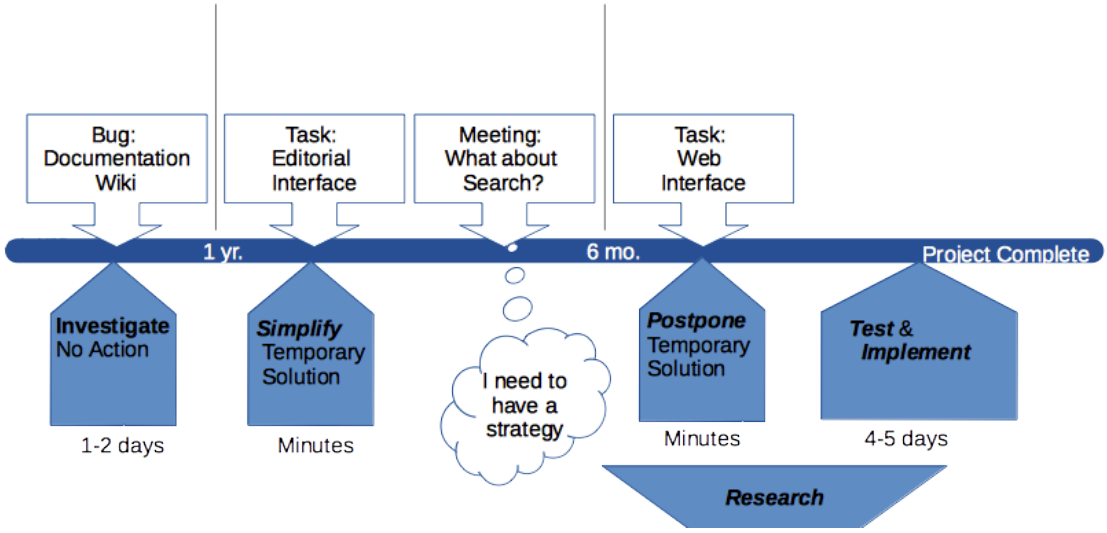


Figure 7.2: Tolerating. Valentin's issue became critical in successive manifestations over a period of a year and a half. Four factors influenced this: the frequency and spread of manifestations, the forms the error took, communication with domain specialists, and a decrease in time to project completion.]

7.3.2.1 This prepared me for that

The issue first occurred in the organisation's documentation wiki early on in the project. A project partner was trying to paste text into the wiki and reported that special characters were not displaying properly. Valentin spent a couple of days investigating, but was not able to resolve the issue. At this stage, the issue was not critical, and a fix was not

required. This occurrence was formative. Within this environment, the text that was added was never intended to be more than a sample “to play with, to experiment with”.

As a result of this occurrence Valentin formed the expectation that he would have subsequent rendering issues. He now knew that the project had a requirement to display a small number of characters from older alphabets that are not readily supported by computers. The investigation also helped him isolate the source of the error as being related to the fonts that are commonly installed on users’ systems. Discussion with a researcher on the team alerted him to the fact that a font exists that supports the display of old English characters.

7.3.2.2 The decision was very quick, and the implementation was very simple

The second time the issue arose, it was more serious. This time, the occurrence was in an editing tool that was being created for the project. The tool was intended to allow researchers from the project to enter and edit text that included special characters. The text needed to be a “faithful reproduction” of the historical material.

With this manifestation, the public dimension of the issue developed. As Valentin noted, for general users on the web it is “not acceptable” to make users go to a different website and download a font. This presents a barrier to access that is considered to be too great. For editors, by contrast, it is acceptable. This is a smaller user base, with whom Valentin has direct contact. He can support them in installing the font.

These factors helped Valentin take a pragmatic decision about how to manage the issue. To do this, he set “aside the complexity” of the problem and implemented a quick, temporary solution. He instructed the webpages to use the old English font, and provided a message directing editors to the page for downloading the font. This tactic allowed him to

focus effort on more important requirements for the project, while analysing the problem "in the background".

7.3.2.3 I had to say something

The third occurrence of the issue was unexpected and marked the point at which it became critical. As Valentin reports, researchers brought up the issue in the context of the search tools that had been created for the site. He was surprised, noting during the interview that he "supposed he had forgotten" what happened when users needed to type special characters.

This occurrence provided two new pieces of information. First, he realised that the problem was more widespread than he had previously thought, as it appeared in a new area of software that he had not considered. It also represented a new form. In this case, the issue was not only one of rendering special characters, but also in supporting readers of the edition who needed to input special characters.

The meeting also clarified how important the issue was to the domain experts. Valentin he found himself during the meeting in a situation in which he was asked a question that he could not answer. He felt pressure at this point to identify a strategy for addressing the issue:

> *"I had to say something, to tell them that I have a strategy, not necessarily a solution, but a strategy."*

7.3.2.4 I wanted to postpone it

The fourth manifestation occurred when Valentin began to develop the site that would display the public edition of the texts. As in the second occurrence, Valentin expected it to happen, and took the decision to instruct the software to assume that the font was installed. As he described it he did this to postpone taking a decision, a tactic he described in two

contexts. First, he described it in in relation to its relative importance within the larger project:

> *"I didn't want to be in the situation where I'm approaching deadline, a phase where we have to do a demonstration or release this on the live website and I have to find a solution in just a very limited time for a problem I've never encountered before. So I'd rather prepare the thinking and explore things in different directions to be sure that I will be ready for that."*

And next in terms of personal knowledge:

> *"I wanted to postpone it so I could work on things I know how to develop and this lets me think about it in the background so I can still analyse things"*

"Analysing things" began with a turn to a colleague with more experience in user interface development. She in turn put Valentin in touch with a second colleague who had still more experience with fonts. The discussion indicated a possible scripting technology that he could use to embed the font in webpages. Valentin had heard of the technology, but had not realised it was robust enough to meet the project requirements.

7.3.2.5 It is never as simple as you explain

The project was nearing completion, and Valentin began to seriously investigate options. One "very ugly" option he considered had been used on another project. It involved splicing images in to spaces between text in order to replace special characters. The solution suggested by his colleague seemed more promising, but Valentin kept the earlier one in mind as a "last resort".

He also began to research the newer option, to ensure that it would be compatible with all of the browsers and operating systems he needed to support. He did this first by performing searches on the internet for information and by "trying things out". This process took several days, and involved him choosing a font, converting it using the

suggested technology and then testing it across different browsers and operating systems. As he described it, the recovery was one of working through "nested problems":

> *"First a problem of Unicode, and special characters, then becomes a problem of browser compatibility, using a technology that you haven't used before."*

He was ultimately confident and adopted the solution. Valentin is pleased, describing it as "very clean" and "well established". He describes himself as "lucky" to have had the help of his colleagues, who helped him avoid accepting an inferior alternative. He is also keenly aware that limits in his knowledge contributed to the issue. As he put it just before our time together ended:

> *"So there is a part of luck and there is also may be related to that the fact that there is a lack of knowledge on my part, and this lack of knowledge could have been different if I had to keep up with what is going on in terms of new development on the client side, the web world. "*

7.3.3 Thrashing

Evan described a day in which things went wrong while setting up a local copy of an open source web application framework. Though relatively new to the framework and to the language in which it is implemented, he did the same task for a different project a couple of months before. That time the process had not been smooth. He had not written anything down, and his goal now was to cement the process of installing the software. He also needed to get the framework installed so that his "real work" for the project could begin.

The task began well. Evan installed a virtual machine running a Linux variant, and checked the framework out of Subversion. He ran the install scripts, then set about getting the framework to run without any error messages, as shown in steps one through three in Figure 7.3.

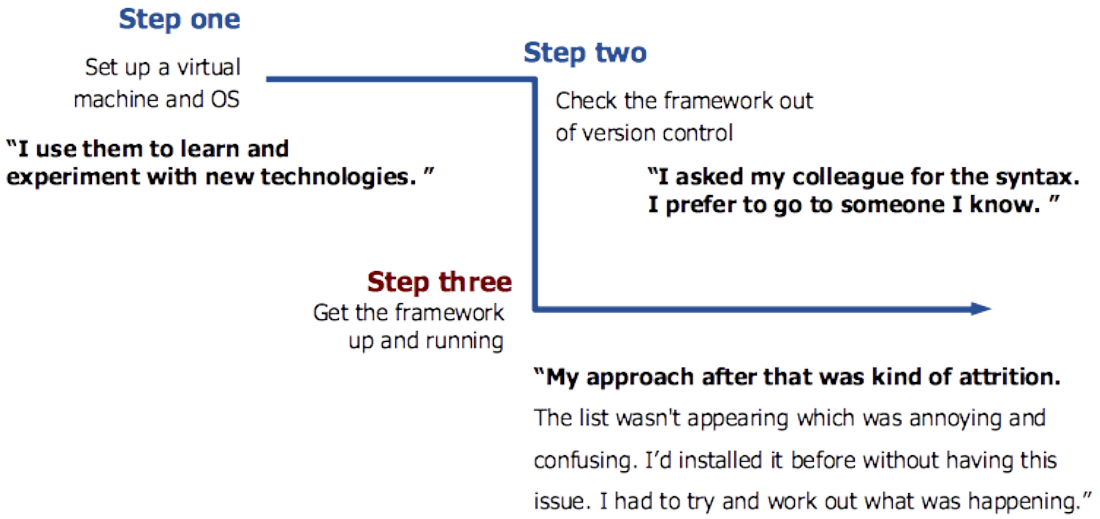


Figure 7.3: Thrashing. The timeline for Evan's incident was compressed, comprising the events of a single day that had taken place the week before. This diagram depicts the initial account he gave.

Once the framework was up and running, he opened the administration interface to create some test pages. He noticed right away that the administration interface looked "a bit odd" because some images were missing. He thought that it might not be a problem, because the page was still functioning, and moved forward to create several test pages. After they were saved, no error messages were displayed. The page indicated with a "graphic or text" that a list should be displayed, however, it was blank. He checked the JavaScript console in his browser, and could see errors related to function calls that were failing because the page hadn't been able to load libraries. He checked that the files were on the system and concluded that "clearly" there was something wrong in a particular configuration file.

It is at this point that the process of "attrition" began. Evan spent a long time looking through the file he thought was wrong, starting and stopping the server, running processes on the database, checking to see that everything was "up to date" and Googling for advice. Eventually he "tracked down" that he was looking in the wrong configuration file altogether. The problem was actually in a second configuration file; in which he had missed or wrongly entered information.

7.3.3.1 What's going on here, why can't you work this out?

Evan resumed testing the framework, relieved that he could finally "get on" with his work.

> *"I added some pages and I had a list of pages, that was good and then I added another page and that was good and then I added a child page and that was bad and it all started to go wrong again."*

Evan quipped in our conversation that at that point he stepped away from the computer and had a cup of tea. This time he had more information to use. The system returned a stack trace and to solve this problem, and he used Google to search for the message that was being returned. He was frustrated with himself, because he knew he had used the module for the previous project. After some time, he found a website that mentioned the problem, which "rang a bell". He remembered that he had had a different problem with the module the first time he installed the framework. He also remembered seeing a page on the department's wiki that described issues a colleague had encountered with the module. Neither issue was exactly the same, but they were close enough to help him identify the module as the source of the problem. To solve it, he downgraded the module to a previous version.

7.3.3.2 Thrashing

Evan shared several practices and preferred ways of working, but he is aware that in this case the approaches often failed. He described the process of locating the configuration error as unsystematic, flawed and risky, and noted at one point an awareness that "if I plugged the dam somewhere it was going to burst somewhere else". The tactic taken to install the latest version of auxiliary software led to the second problem, which left him frustrated and confused.

Though he prefers to ask colleagues for help, on the day of the incident, Evan was working at home, tunnelling into a virtual machine hosted on his machine in the office. The need to switch between environments on a small laptop screen confused him and he

became turned around about what he had done, "what I'd changed and what hadn't changed.". In the end it was information from the internet that helped him identify what the problem was.

The way Evan describes the day suggests that the issue was not necessarily critical, but it was severe. It is likely that he was thrashing, that is, that he got lost during problem solving and that the experience was stressful. Evan considers his experience to have been a "personal failure", but also useful. It forced him to take a closer look at the software he was using and building. As he put it:

> *"You know this is quite informative 'cause obviously you would get something and it would work out of the box and you don't really think about [it] again, so even though this was an annoyance, it was quite useful to actually have to look into those relationships."*

He reflected that his knowledge of the application framework had grown as a result of using it on two projects:

> *"I'm comfortable with creating that environment, I'm comfortable with getting up and running and also I'm much more aware of creating something that's got a bit of longevity."*

At the time of the interview, Evan believed that everything was working, "touch wood". His confidence was not high; he expected that more issues would come up when he promoted the code to the next environment.

7.3.4 Piecing

Robert described taking over a task for a colleague that had gone on holiday. The task involved performing client-side validation on portions of a form that were dynamically loaded by the server based on actions taken by the user. The form validation worked in when the entire form was loaded into a page at one time. During testing, it became clear

that the validation was not working properly when the interaction was more complex and parts of the form were loaded at different times.

7.3.4.1 The answers steered me in the right direction

This was a standard issue, and so Robert duly turned to the internet for guidance. He was quickly able to determine that the issue involved a JavaScript library that was being called into the webpage. He discovered that there were multiple questions that had been posted that were related to the issue he was having and "lots of advice" about how others had solved similar problems. He took what he found, tried a few things in the code, read "more and more" on the internet, and tried to implement a couple of solutions he found posted.

One post suggested that if he unloaded all of the elements on the page from a component in a library, and then reloaded all of them, the client-side validation would bind to the newly revealed fields in the form. This made sense to Robert, who felt that what he was trying to do with the library was not "too far out of the ordinary". When he tried doing this, however, the validation still did not work.

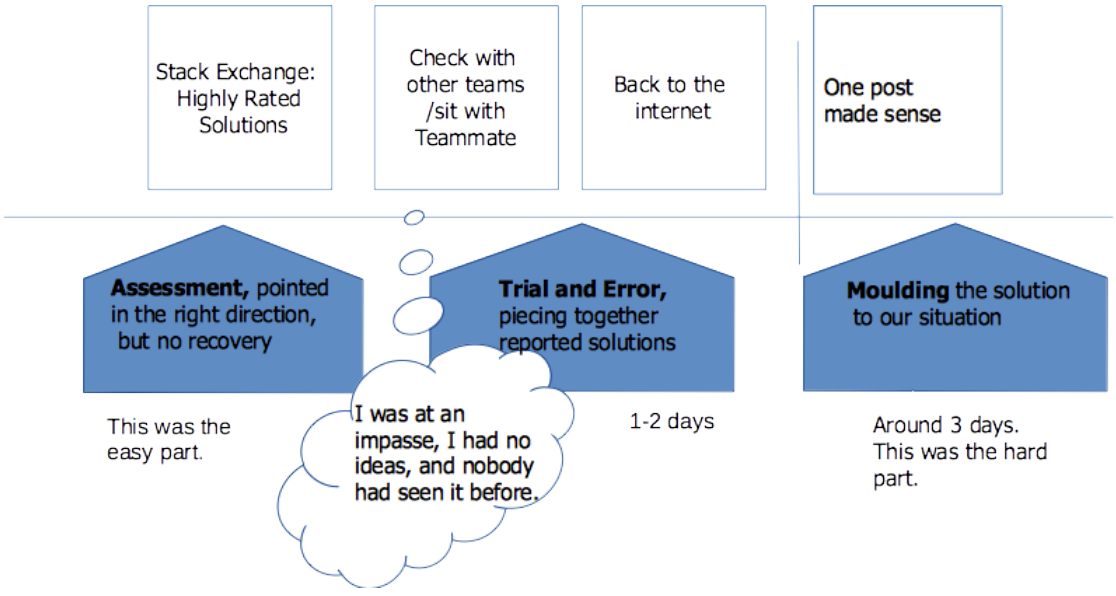


Figure 7.4: Piecing Together. Robert took responsibility during a sprint for a task when a team member left for holiday. The problem was detected while testing edge cases in user behaviour.

7.3.4.2 It wasn't something they'd done before; it wasn't an issue they'd come across

At an impasse, Robert found a team member who was willing to talk through the problem with him, but time constraints meant they did not get very far, so he turned to other team members and then to other colleagues in the department. He expected that by speaking with people who had more knowledge in the area he might find someone who could look at it, “know enough and provide an answer straight away”. At a certain point, he realised that he would not find help in-house, so he turned back to the internet.

7.3.4.3 So it is kind of like a double check

The model for validating the form is static. First, the client-side validation is supposed to check entries to the form before it is sent to the server. The checks are for “simple stuff”, to make sure that mandatory fields have been filled in, and that data is in the correct format. On the server, the data from the form will be validated again. First, the same checks that were done on the client will be repeated “just in case” and then cross-field validations will be performed.

The interaction model that produces the form to be validated is dynamic. Some form fields are hidden and only revealed based on actions taken by the user. As more parts of the form are loaded, the client-side validation needs to bind to the new fields.

Robert discovered that there was an error in validating the form when it had been loaded dynamically. Simple interactions worked as expected: when the entire page was edited and saved, the client-side validation triggered properly. However, when the form was saved after entering no data, the client-side validation did not occur. Instead, the form was sent to the server and the server-side validation fired.

7.3.4.4 Piecing Together

Robert achieved a fix only once he was able understand the solutions that he found on the internet "enough", a process he describes as "piecing together", and which is depicted in Figure 7.4. He turned back to internet sources, which he studied "quite a lot", and found one highly rated post that explained in detail a method for extending the JavaScript library he was trying to use. He found the fact that the poster could extend the library useful, and he was able to see some sense in how the extension they described would help the poster meet the described goals.

As in Evan's case, the details of the information found on the internet did not exactly match Robert's situation. He did not think he would ever have the same use for the extension. It was how the poster had explained what they'd done and the timeliness of the information that helped him understand what he needed to do. This marked the end of the "initial hard part". Then began the hard part: implementing the solution and "really working all the way through it and moulding it to what we were doing".

The heart of the issue for Robert was that it was "new to me". Recovery depended on his ability to find a solution by determining the "right" thing to look for on the internet and then learning enough to recognise the right answer when he saw it. As demonstrated in Chapter 6, the recovery was not accompanied with full understanding of why it worked. Robert was comfortable with this, noting:

> *"If I was going to go back and approach the issue again, it would be a case of trying to make sure that I did understand what was going on in the framework upfront, but there is so much to know that you just need to make sure that you understand enough to make it work at this point in time."*

7.3.5 Naming

Thomas described a task he took on to implement a series of forms on a website. He has worked with web technologies in the past, but is less experienced at user interface

development. Because of this, he began the task by arranging a meeting with a more experienced colleague. That person explained how they had approached writing and organising a set of pages with similar behaviour. When Thomas came back to the task the following week, he met with a different team member who would share some of the work and started to explain the task that had to be done.

7.3.5.1 You've got to remember where you are

The interaction model Thomas needed to support is complex: users can expand and contract different areas of screens, and elect to edit the finer detail of individual elements. When a user decides to edit information, they are directed to a new page. After saving edits, they are returned to the starting screen. In this case, the layout of the original screen must visually indicate their last point of focus by expanding the area of the form that they were looking at when they navigated away.

Thomas thinks that some of the challenges of supporting interaction are related to changes that "came in" with web development. As he described it, managing interaction in desktop applications was comparatively simple. On a single computer, information about where a user is in an application and the actions they take is "nice and easy to store". On the web, by contrast, each page is individual and information has to be held within it:

> *"And then cause there's other pages that go off and how far you have to pass that over and how many times, but then also when you come back, what needs to be displayed when you come back?"*

7.3.5.2 It was making it clear about what the names were being used for

Explaining the task to his co-worker did not go well. The problem was in the naming. Thomas knew he had a variable that had been defined in the class structure on the server that related to differently named parameters on two web pages. The colleague asked why there should be different names in different places. Thomas struggled to answer and began

to wonder if he was putting values in the right places in the structure he was describing. The only answer he could find to give his co-worker was that it had been done that way before. He realised “I haven't quite got this right.”

He explained that moving between screens is not hard to manage technically. The team is using a piece of mapping software on the server that can map differently named variables to one another. It takes some setting up, but then it glues components in different areas of the stack to one another. The difficulty is in managing the concepts that relate to one another on different screens. The interaction path can involve visits to several different pages. An ID on one page is used for display, but if the user navigates to edit a portion of the page, it becomes a parent ID. The related parameters could be given the same name on each page, but this isn't desirable because it doesn't indicate what the parameter is being used for on the current page.

The global aim related to naming parameters is to support future developers. It was important to make clear by the names what the parameters were being used for and how they related to names given on other pages. A second aim was to choose names that were consistent with choices that had been made in the past.

7.3.5.3 I need to make sure I've got this right.

Thomas spent between fifteen and thirty minutes trying to “explain on” the naming of the parameters based on what he'd been told but his colleague did not understand. Thomas recovered by bringing the original person back into the meeting, and the three began to draw on a whiteboard. They blocked out the screens and intended interactions, and sketched the pattern the previous developer had used.

At a certain point, Thomas thought that his other team members might benefit from hearing the discussion, so the entire team was brought in. His thinking was that if he was

struggling to explain the concepts to someone else after having learned them, it would be even harder to explain it again in the future.

With everyone in the room, discussion expanded to consider different ways to manage state for individual elements on the pages. Someone brought up what should be placed within the parameters. Reference was made to work performed several sprints ago to change the way page navigation was tracked through a user's session. Some problems were noted for “future development”. The time at the whiteboard was not recorded. The understanding given was that the discussion would be sufficient to provide foundational information to the team.

7.3.5.4 Even if you don't pick it up in six months, you are aware

Thomas believes that a “mixture of things” aided recovery. Working with diagrams made a “big difference” as did talking it out, discussing it as a team. Having to explain the problem was key, because it revealed the gaps in his understanding. When asked, Thomas noted that the naming issue turned out to be the simpler problem to solve. That one was easy because it was possible to get the original person in, and they knew what had to be done and why. They could explain it a second time.

The hard part emerged during the broader discussion with the team. This issue was different from others he has encountered while working on this team. Usually he has found that problems are “quite small” or centre around differences in opinion about how things should be done, or about how best to accommodate decisions that have been taken “higher up the tree”. This time, Thomas had to “stop and think”. It was a problem he “didn't have an out and out answer for straight off. It had to be discussed.”

7.3.6 Slipping

Dereck described the impact and consequences of a slip made while building and deploying software. It demonstrates an issue for which recovery was achieved by abandoning a

fix. Dereck's team maintains two curriculum services for the department. One provides a list of course names, the other a list of IDs. One day Dereck made a change to the ID service so that it would reference the production database rather than the maintenance database. It was a simple change to make. The code was built and deployed to the server, and Dereck left for lunch.

When he came back, he saw a lead from another team talking with his colleagues. The team lead was reporting that the course name service was not working. It had just stopped, but no error messages were being given. Dereck was surprised, because he had not made changes to the service that was being reported as down. He checked to see if anyone else had changed that service in the version control system or if it had been deployed but it had not. He wondered if the other team had done something wrong on their side because when he tried calling the service locally, everything worked. Then he logged on to the web server to check the files, and realised what was wrong.

The deployment of the ID service requires a manual operation to copy files into the proper directory on the server. The manual step had been performed but the files had mistakenly been copied into the naming service directory. The naming service was not working because it had been overwritten. It should have been straightforward to restore the overwritten code, but he could not find it in the recycle bin on the server. It had not been deployed for several months, and so backups had been cleared out by a date-based automatic process.

7.3.6.1 It was probably me

"[W]e all do deployments. In all fairness, it was probably me that made the error." When Dereck saw that the files had been overwritten, he knew that "human error" was the source of the problem and he knew that he was responsible.

Dereck's slip was critical. It caused his team to break their contract to other teams in the department. The software in the department is all under development, and individual components are in varying degrees of stability. Teams have dependencies on one another, and there is the risk that if one team makes changes it will break something for another team. This has been a problem in recent months, and a policy has been set that teams responsible for services have to put something in place when maintenance occurs to make sure that services never go offline.

7.3.6.2 On a headhunt, trying to work it out

Dereck discussed several alternatives for recovery. First, as noted, he might have restored the service from a back-up on the server, an option that was not possible because the service had been dormant for a while, and backups had been erased.

Second, the code for both services could have been redeployed, and the manual step could have been properly performed. This was not possible because changes had been made while refactoring the ID service that impacted the naming service. Deploying the naming service would have resulted in a different failure.

Ultimately Dereck determined that the best thing to do would be to alter the build to deploy the older version of the broken service. The diagnosis and identification process finished at about four o'clock. Afterward, Dereck spent several hours "on a headhunt" trying to work out how to alter the build. To do this, he searched the web and within videos and documents provided by an on-line training service that the department subscribes to. He could not find anything that would help him. The situation Dereck worked through is diagrammed in Figure 7.5.

7.3.6.3 We end up having to develop workarounds

The challenge he faced in recovery had to do with decisions that had been taken about how the software had been structured, and how that structure was related to the code repository

and to build and release management. The two services were implemented within a single file, and so making changes to one stands to impact the other service. When the code is checked in, the file is given a new revision, which is applied to both services. By contrast, the service architecture is configured so that the services are independently represented, and the build is configured so that each service is independently deployed. Normally this means that one service can be deployed out to the web server just by running the build. The other service, however, has to be manually copied up.

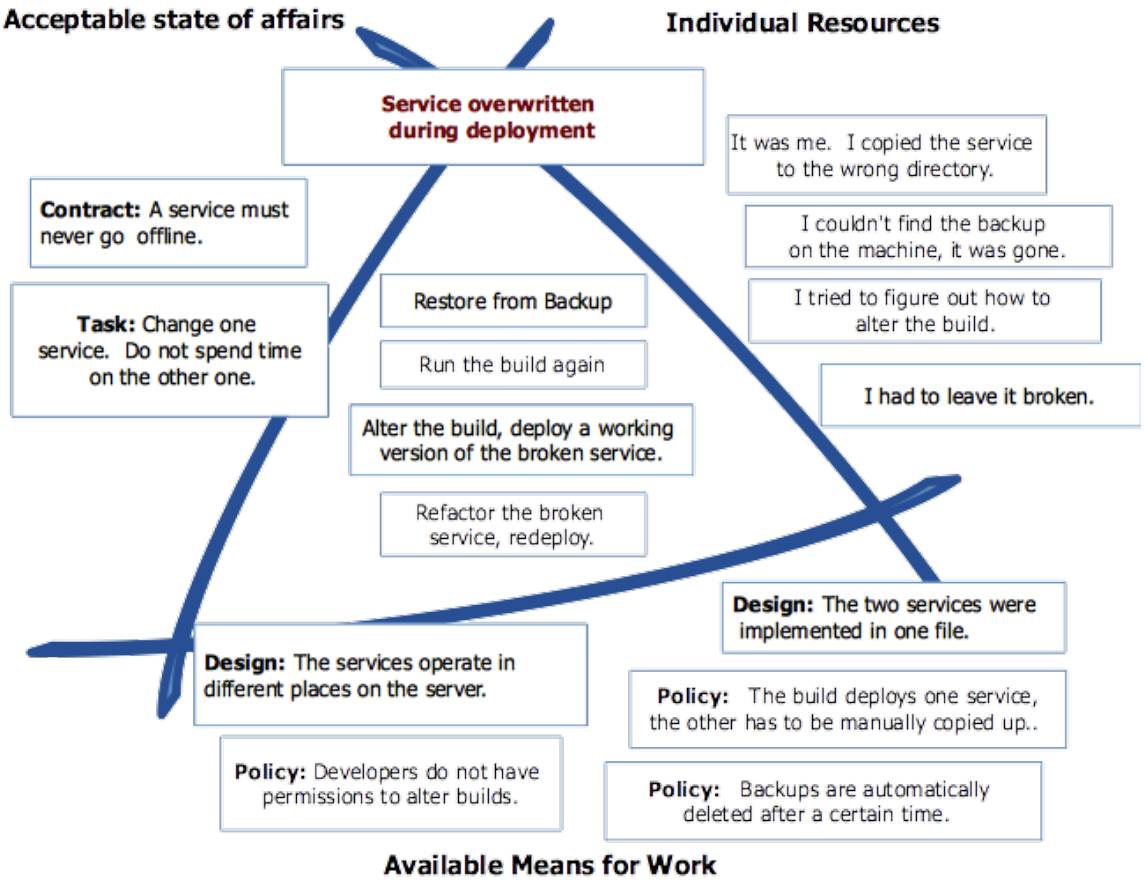


Figure 7.5: Dereck's Slip. Four options were considered, but Dereck was not able to recover from his slip. In the end, the task was abandoned, and all of the changes were rolled back. Diagram adapted from (Rasmussen, Pejtersen, & Schmidt, 1990).

Dereck lamented that the architectural choices mean that the team has to "work around something else and something else", but this configuration has also given the team

freedom in how they organise work. The two services share code, and so the task might have been set to refactor both services to point to the live database. The email Dereck sent to his lead suggests that he explored this option, but it was not taken in the end. Dereck indicated in the interview that this option was not desirable, because the team had only "tasked a story and estimated doing the ID service". He explained that the naming service was likely going to be dropped in the future, and so the team had specifically been told not to spend time on it, to leave it alone.

7.3.6.4 None of us knew at that point in time

In the end, Dereck had to give up. He felt "quite down" on himself for leaving things broken, but the roll back seemed to be impossible. He sent an email to his team lead explaining the situation. As he was not due to be into the office the next day, his team-mates would have to solve the problem, when people in the department who had permissions to alter the build would be in the office.

He thought he would come into the office on Monday and "it would just all be sorted" but no one on the team had been able to work out how to roll back just one of the two services. The task for the sprint was abandoned, all the changes that had been made were rolled back and both services were redeployed. This restored the contract, and though both services were again using the maintenance database, the other teams were "none the wiser".

7.3.6.5 We just think this is good design

The team knew that their domain model was stable, and so the issue has had farther reaching consequences. As it turns out, access to databases is managed in multiple libraries. Now that the team feels more confident with the domain model, they are going to consolidate access, meaning that configuring switches between live and maintenance

databases will only happen in one place, and will not need to be done within the code for the ID and naming services.

This change will not remove the possibility that Dereck's slip will happen again. Going forward, the services will still be independently deployed, and the manual copy to the web server will still need to be done. Dereck had this to say about that point:

> *"If you want the truth, we could have worked around this a long time ago… [M]aybe we've kept it this way, to highlight that there is an issue and so that when situations occur like what we've gotten into we can say well, in all fairness…"*

7.4 Discussion

The lapse of time that passes in software development between actions and outcomes is a known challenge in bug fixing (Eisenstadt, 1997). Likewise, developers are responsible for their own actions and must "believe" in those taken by others (Naur & Randell, 1969). Software developers rely on an ever expanding array of software written by other folks, and belief is a factor that has subsequently grown to have greater force in professional contexts. Belief also has a second dimension. Errors in professional software development are frequently detected and reported by other people: by testers, by users, by clients or colleagues.

This section discusses how developers respond to outside influences in practice. The first section describes in general terms the nature of tasks that require ***conscious*** problem solving. The next section expands the concept of ***suspicion*** discussed in Chapter 5 to include ***responsibility***. The last section considers how developers describe and use rules of thumb.

7.4.1 The Nature of Tasks

In the course of conversation, Robert shared a taxonomy he has heard developers used to identify tasks. Robert explained that problems in software development could be categorised as:

- Things that we've done before
- Things that we can Google an answer for
- Things that no one else has done before

Though given in the context of his own experience, the taxonomy is notable for three reasons. First, the points are representative of the kinds of problems reported at both field sites. Second, the levels of the taxonomy can be associated with the levels of conscious handling required for different actions described within psychology by Norman and Reason, and of the levels of performance regulation observed by safety science researchers. Drawing comparisons between Robert's categories and the typologies summarised in Chapter 2, Section 2.2:

- Things that we've done before are tasks that are routinely performed, or well-learned.
- Things that we can Google an answer for are problems that are recognisable and can be solved using prior experience or through shared "know-how".
- Things that no one else has done before are unfamiliar, or novel tasks that will require local goals to be explicitly set, undertaken and evaluated.

Finally, the full account given by Robert demonstrates that it is not straightforward to assign details of professional performance and experience to fixed categories. When asked to categorise his own incident using the taxonomy, Robert immediately placed it into the second category. The task was "pretty standard" and was something that could be Googled. However, it involved doing something "slightly different to normal". When asked, Robert explained that the slight differences in the combination of client and server side frame-

works caused the issue to "touch into" the third category of the taxonomy. Robert didn't understand the frameworks, he had never done a task like this before, and neither had anyone else in his group. So it was Google-able, but it was not the sort of issue for which an immediate transferable answer could be found.

Robert, like the other developers who were interviewed, is experienced, but the account he gave was of an issue that was "new to me". ***Newness*** is one of the "special conditions" that triggers the need for conscious handling (Norman & Shallice, 1986). The notions of novelty and of the timely need for knowledge marked all of the interviews. Dereck explained that blame could have been placed on tools or knowledge but that he thought the main issue was one of timeliness. No one on the team knew what to do ***at that point***, and so the path to recovery taken was of necessity "tactical".

Knowledge is required at particular points, but software developers are aware that the passage of time is key to its development. Dereck's team took a provisional approach to recovery with an awareness that the problem remained active. The factor of time also influences individual behaviour as demonstrated by Valentin who allowed more than one bug to surface over the course of months, while he explored alternative solutions and solicited feedback about priorities and requirements. Joachim described that his knowledge of the library he was using was still forming, and the larger chronology of his issue revealed that the bug he wrote was of little importance, one knot in a longer thread of practice.

7.4.2 The Need to Witness

Errors that are reported by others are described in the error detection literature as not happening very frequently (Zapf & Frese, 1994), but the importance of other sets of eyes is also described as being necessary for diagnoses of higher-level "knowledge-based" errors in critical, dynamic work environments (Woods et al., 1994). Reports of error are, in one

sense, the bread and butter of software development on the job. Software developers must, as Marcus and Joe did in Chapter 6, regularly assess reports given by system responses, but they also must assess reports given by colleagues or within bug reports.

Taking **responsibility for an error** is a big deal professional software development (Guo et al., 2011). There is the possibility that a report is untrue, or that something was done wrong by someone else. It may be the case that the report is incomplete or represents a misunderstanding on the part of the reporter. Recreating reported behaviour is a standard tactic to employ in debugging tasks (Lawrance et al., 2013).

Recreating reported behaviour develops ***awareness*** and predicates error handling. Awareness is a two-fold notion. A person must realise both that something is wrong and that one is responsible for the error (Rizzo et al., 1995). Awareness is at times instantaneous ("I poured the coffee into the sugar pot!"), but can also emerge after time has passed. The presence of an error is sometimes established independently of the notion that one is personally responsible for it (Rizzo & Bagnara, 1995). A person may observe outcomes of an erroneous action before they associate the effects with something they did.

A developer may begin investigation of reports by assuming (or hoping) that the error has nothing to do with actions he has taken, that he is not responsible for an outcome. He may deflect responsibility by making the problem space big ("It must be a memory handling issue in the browser!") or by setting a boundary ("I've checked everything on my side…Maybe they've broken it on their end"). This is a variation on setting constraints observed in the design session with Kasia and Bill who used the tactic to focus activity on the present moment. Deferring responsibility can be used to identify missing and incorrect information in reports as Joachim did, to buy time to think as Valentin did, or to direct investigation as Dereck described, by looking at factors in the local environment before expanding investigation to the server.

7.4.3 Rules of practice

Rules are declared and observable. They are described by developers in terms of things they like or do not like to do, or as "instructions" they follow. They were often also relayed within descriptions of technical knowledge, for example "basically you have to…". Sometimes these kinds of statements were used by informants to demonstrate technical prowess or to prove vitality on a team or within a department. However, they just as often revealed personal rules-of-thumb (Rasmussen, 1985) a developer used to manage practice.

Evan's story included several accounts of preferred or learned practice, summarised in the table given in Table 7.5, below.

First Principles	Working from the assumption that I know nothing by throwing myself into a task, and figuring things out.
To Learn and Experiment	Specific tools like visualisation tools are used to provide a space to learn and to mess up.
I Prefer to Go to Someone I Know	Seeking help from colleagues is preferred over internet-based sources which can be "deliberately obtuse"
See What Gets Spat Out	An error-driven practice to manage software installation. The steps to follow are taken from system responses.
Always Pick the Latest	When installing dependent modules, use the practice of installing the latest version first.

Table 7.5: Evan's preferred practices.

Conventions guide and direct practice, they can make work easier. For example, Evan described using **See What Gets Spat Out** in this way:

> *"[T]here were modules that the application referred to in the settings that I didn't have installed, so. (pause) Go off, get them, install them, and try again, move on to the next error, work through that. "*

Rules reveal learned behaviour, but also aspects of how developers reason in particular situations. In a moment of reflection, Evan noted that at a certain point in the investigation, he had reached the limits of his knowledge and experience. The description he gives to "get it back to how it was" sounds in the telling like a practice, but the evidence suggests that the actions were tactical, firmly embedded in the situation:

> *"I'd spent long enough messing with the configuration files. I realised either it wasn't there or I'd broken it completely. Let's get it back to how it was - you know I think you take a step back and you think okay it should be working the way it is so let's move on to the next thing and try and understand."*

It is important to note that conventions of practice are fluid. They are not fixed or uniformly helpful. **Always Pick the Latest** is a practice that has worked for Evan in the past, but results this time in an unexpected error that must be handled. Rules may not serve in a current situation, and may even result in bigger problems. It is also important to note that just as they form and reform in new situations, they are not always followed. Robert described that he sends an email explaining progress with particularly tricky issues to team members, but found when asked to forward the email to the researcher for analysis that he had not followed the practice this time.

7.4.4 Limitations

The stories given in this chapter situate error within time and organisational context. Stories were collected from two organisations, and may not represent software development in different sectors, or in organisations with different work practices. Accounts were gathered retrospectively and it is possible that details were forgotten or distorted. As might be expected, finer detail was collected about activities that occurred close to the point of interview. Finer detail was also collected about activities at the computer when the developer was able to recreate and demonstrate actions during their account.

In general, the interviews collected at Site B provided much more detail about how local problem solving was performed than those taken at Site D. This is likely due to the fact that developers at Site B were interviewed at their desks, while developers at Site D were interviewed in public spaces and had to contextualise their discussion in relation to hastily created diagrams.

Researchers must be careful when making inferences about accounts of process. Informants may present a view that does not reflect what was actually done (Hammersley, 2003). There were clues given that sometimes the informant recounted a desired practice rather than what he actually did. It was also the case that developers sometimes "hid" accounts within more general explanations of technical or organisation process.

In general, however, and as Eisenstadt noted (1993), informants were forthcoming and generous in sharing experiences, and gave no reason to "distrust". Detailed analysis of the accounts revealed self-consistency: it became apparent what the account was and where and how informants postured or obscured detail.

7.4.4.1 Bugs and War Stories

Bugs featured in some of the accounts given at Site B. This may have been due to the way in which the research was described in the information sheet given to informants (see also appendix D.4). The starting point for discussion with Joachim was a bug that had been reported to him by a project partner. Valentin, despite describing his issue as “not necessarily a bug, it’s an improvement” recounted that his issue nevertheless manifested as a bug four times in the course of a year and a half, in different pieces of software that were being used and built. Evan casually referred at one point to one of the issues he encountered as a “bug” that was like one he had encountered in the past.

Bugs did not feature in discussion at Site D. Instead the stories were recounted in terms of tasks that had been set for a sprint. All of the informants spoke of testing in relation to

their work, making reference to performing unit testing, detecting a problem while testing "edge cases" and in using the testing framework to see of code was running locally.

Two war stories were reported (Orr, 1986), one from Site B, and one from Site D. James described having a terrible time making use of an API supplied by the Eclipse development environment. His difficulties arose after the Eclipse developers had changed behaviour in APIs used by the public. Dereck related a story from a previous job in which a colleague had "dropped a clanger" when installing an update to server software.

These stories were not included in the error handling analysis because they did not meet criteria for incidents: they were presented as anecdotes, and did not give sufficient evidence of narrating or "summing" up in the midst of the experience. However, both did yield information that contextualised error handling during software development. Dereck's account helped define issues that are critical because they are visible to people outside of individual experience, while James' account gave insight into the ways developers seek guidance from internet sources.

7.5 Conclusion

This chapter reported a qualitative study undertaken to examine how developers recount problems they have solved in recent work. It reported the activities of six developers working in two organisations.

Errors can illuminate aspects of individual cultures of development, software engineering practices, or model of design in which they arise (Curtis et al., 1988). Occurrences provide feedback about the nature of problems in specific domains. Errors come to developers in one of two ways. They may ***come down*** as a result of a task that is taken from someone else or problems reported by clients and co-workers. Issues may also ***come out*** of actions taken by developers. In both cases, responsibility for the error may only be taken once the error can be witnessed and linked to prior activity.

Several of the accounts capture the complexity of the relationship between local and global problem solving in software development. Problem solving in software engineering is often described in terms of large, global aims: commercial strategies, project requirements or of design decisions. By contrast, findings in this study support the view of error handling in psychology and safety science findings that problem solving is often local and small, it can require cycles of practice that blend skill, experience and reasoning.

The knowledge required in software development is timely. Robert described how recovery depends on learning enough right now to ***piece together*** a solution . Joachim's experiences showed that errors sometimes form knots within longer ***threads*** of development practice, threads that relate both to individual and global aims. Valentin likewise ***tolerated*** the error he encountered, allowing it to reoccur more than once, in order to gather feedback from technologies and users.

Workers must translate work goals into personal tasks that can be undertaken (Frese & Zapf, 1994), and software developers must consider global aims while managing local problems. In the context of error handling, this translation process has a large influence on personal development. Evan's story provided rich perspective about how developers gain experience by ***thrashing*** toward solutions. Thomas caught his error ***in the act*** of explaining how elements on different web pages relate to one another. He recovered by giving the error back to the person who had originally explained a practice to him. Though the consequences of errors are generally depicted in terms of the ill effects a bad choice in programming has on software, Dereck's ***slip*** starkly demonstrates the effects that organisational policies can have on individual experience.

8. Discussion

The previous chapters explored error in the context of design, at the desk, and after work had been completed. Findings illustrated different features of error handling within these contexts, exploring how developers undertake problem solving when things go wrong. A sense has been given of how developers perceive problems, the sources of information they draw upon during handling, and how handling unfolds in-time and over time.

The problems developers face generally aren't new, they are "new to me". In other hands or at other points in time, tasks may have been or become routine or mechanical, but at the point that handling is required they are novel. This is the "special condition" that most often commanded attention (Norman & Shallice, 1986, pp. 2, 8). Likewise, error handling may include elements of understanding or of use but was largely required in these accounts to manage errors in ***making.*** This distinction broadly characterises incidents across the sets. All of the developers were tasked with making software, and encountered problems in the effort.

This chapter draws out subtler distinctions of error handling in software development practice. Data drawn from the studies reported in Chapters 5, 6 and 7 is situated within the theoretical framework presented in Chapter 3.

8.1 Characteristics of Handling

It is, by now, a familiar refrain. Error handling unfolds in three stages. It begins with detection, with knowing that something is wrong. Once an error has been detected, a developer must identify what was done wrong and what should have been done. He must take steps to remove the effects of the error. This process was given a theoretical overview in Chapter 3, Section 3.3 and was described in the context of desk work in Chapter 6, Section 6.4.

Figure 8.1 below depicts the three stages of error handling that have been identified by studies in psychology. Detection indicates that someone realises that something is wrong, identification is the process of knowing what should have been done. Effects are removed in recovery. The examples given in studies of error detection suggest that handling for simple errors is straightforward and brief. Insight is unambiguous; identification and recovery are more or less instantaneous once a detection has been made. Findings support this; some errors, like the one depicted in Figure 8.1 below could be described as slips of action and are relatively easy to handle.

However, the accounts analysed in this research suggest that error handling in software development is influenced to a great deal by environment and circumstance. The process often involves more than one kind of performance and may include multiple errors of different kinds. A diagram including the factor of time and the notion of local problem solving is given in Figure 8.2, below. This is consistent with broader descriptions of workplace performance, in which skill- or action-based errors tend to precede the detection that something has gone wrong, while rule- and knowledge-based mistakes arise in the subsequent efforts made to solve a problem (Reason, 1990, p. 56).

In the following three sections, characteristics of handling that relate to the three stages of detection, identification and recovery are described. Discussion is grouped into the descriptive categories coined by Sellen in 1994, first described in this thesis within Chapter 3, Section 3.3, and discussed in Chapter 6, Section 6.4.

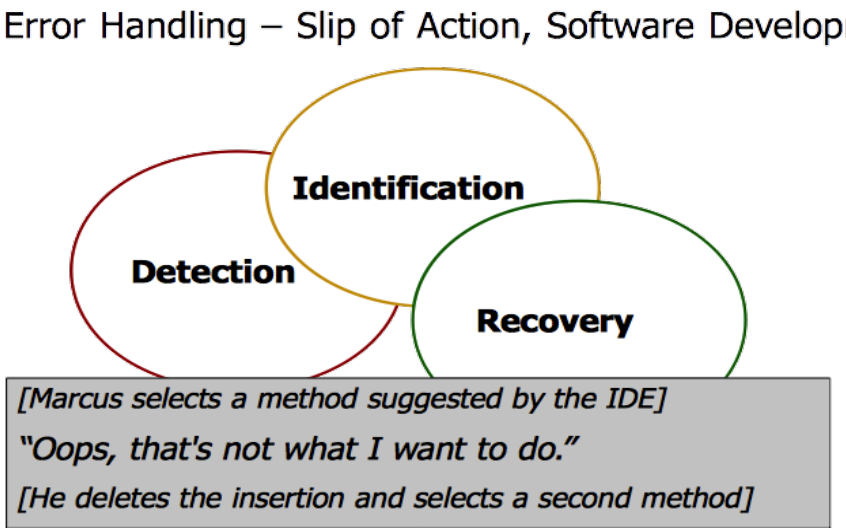


Figure 8.1: Error handling - Slip of action, software development. This is a brief handling incident for an action-based error, like the ones depicted in related studies reviewed in Chapter 3, Section 3.3. This particular incident is described in more detail in Chapter 6, Section 6.3.1, and in Appendix C.3.1.

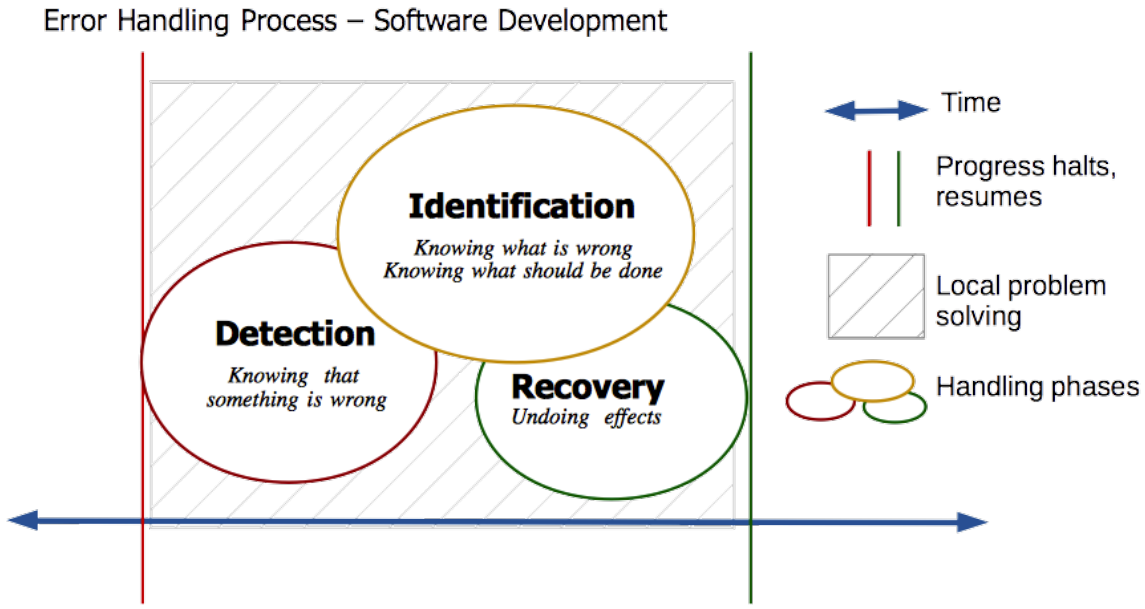


Figure 8.2. Error Handling Process - Software Development. This diagram situates the basic phases of handling that were introduced in figure 3.4 within a time frame, indicated with a blue bi-directional arrow, and two coloured bars in red and green that indicate points at which error handling replaces progressive problem solving. The grey hatched box behind the coloured handling bubbles indicates that error handling instances can involve the need for conscious, local problem solving

8.1.1 Detection: Knowing that something is wrong

Actions sometimes do not go as planned, or were not intended (Norman, 1981). They are often simple, routine or skill-based, and are commonly detected based on perceptions that

arise while ***doing something*** (Sellen, 1994). Copying files from one place to another is one example of a simple action that is routinely performed in software development. Copying files into the wrong directory, as Dereck did (see Ch.7, Section 7.3.6) is an action-based error. It could be classified as a ***slip of action*** within Norman's typology.

Determining what constitutes "in the act" is not always straightforward. A developer may be able to swiftly assess what he meant to do, compare it to what he did, and take subsequent actions to correct their input. Thus, though an error may seem to have been detected in the act, developers likely are often also assessing small outcomes along the way (Sellen, 1994).

As demonstrated in examples of error at the desk given in Chapter 6, an error may be swiftly handled because it is familiar, having been seen and managed before. These are quibbles. Some errors are quickly detected, quickly enough to have been caught in the act, as depicted in Figure 8.1, above.

Even in the case of slips, detection of errors does not always occur immediately. Sometimes, as in Dereck's case, detection is delayed, made after an action is taken and outcomes can be assessed (Sellen, 1994). Outcome-based detections may be reported or self-detected, arising as a part of a standard check or out of a sense or suspicion that an error has occurred in recently completed work (Allwood, 1984).

Errors may also be detected due to a failure to identify what is to be done. Lost intentions are typically termed **lapses**, and described in terms of memory (Reason, 1990). Forgetting why one has opened a file in the middle of a development task could be interpreted as a lapse. However, at times, people detect an error because they realise that they ***do not know what to do*** (Zapf & Frese, 1994).

Handling often unfolds in terms of whether a detection is made using extrinsic or intrinsic information. Characteristics of both sources of information are discussed in the next two sections.

8.1.1.1 Extrinsic information

Errors are often signalled through "evident" information that would be apparent to any observer (Zapf, Maier, Rappensperger, & Irmer, 1994): through system responses in the form of red bars, stack traces or other messages that either gag the system or provide warnings about an error condition (Lewis & Norman, 1987). In this case, detection is more or less guaranteed (Reason, 1990), spurred by something designed into the objects of use that force (Norman, 1981) or limit (Sellen, 1994) forward action.

It was argued in Chapter 6 that software development at the desk is ***error-driven***. Methods like test-driven development are designed around failure (Ambler, 2012). Developers adapt their practice in response to messages given in development environments and come to depend on information from system responses to direct and manage activity at the desk.

Error-driven practices are observable and reported across the data sets. Evan expected to see errors that "gag" the software during his software installation. He *relied* on them to direct his installation process. In so doing, system responses replaced other intrinsic or extrinsic sources of information he might have used such as memory, notes or documentation. Marcus and Joe provided other examples of using errors to direct practice. Their aims were methodological, as when they wrote tests to fail, but also personal, such as when they left errors behind within the IDE to serve as placeholders for picking up work at a later time. Findings in the studies support all of these points. Robert detected his error while performing standard checks on system response.

Developers respond to system responses, but are also responsible for writing and designing them (Lawrance et al., 2013). System responses are thus also leveraged, elicited in completed work as a checking mechanism (See Robert's account in Chapter 7, Section 7.3.4). Robert described looking for an intended system response to evaluate changes that had been made to code. He was using errors to confirm that the model for providing validation sufficiently corresponded to the possibilities for interaction. His error was detected on the basis of an *untimely* system response that indicated an aberration in the expected sequence of validation checks.

Errors are sometimes detected when things do not "look right". Developers use visual sense to gauge whether things are working or not. Messages that the developers have written into system responses may lack or misinterpret information, cueing detection. It is *the way* that an error message is formatted that signals to the developers that something has gone wrong. A character may be interpreted as syntax when parsed by the browser, or messages may not contain information that developers have designed to be included.

Similarly, systems provide responses to developers that have not been directly designed to support error handling, but which are understood by developers to be indicators of problems. Error detection in these cases depends on the developer understanding how software is meant to "look". In such cases, a user interface may display all of the intended textual information, but lack visual elements such as images, spacing and fonts. Evan expected the software to work correctly once the software ran without any errors, when it was no longer gagging. However, he noticed that things did not "look right", cueing a suspicion in him that an error had occurred. In fact, the web page he loaded was not rendering correctly. Images were missing, and pages that he created through the interface were not being displayed in lists.

In a similar incident, Marcus recognised that something did not "look right" in the web-output. The page loaded with all of the expected information, however the visual style of the page was incorrect. The top figure below, 8.3 displays what the screen looked like when something was "wrong", the bottom displays the "right" look. (Figure 8.4).

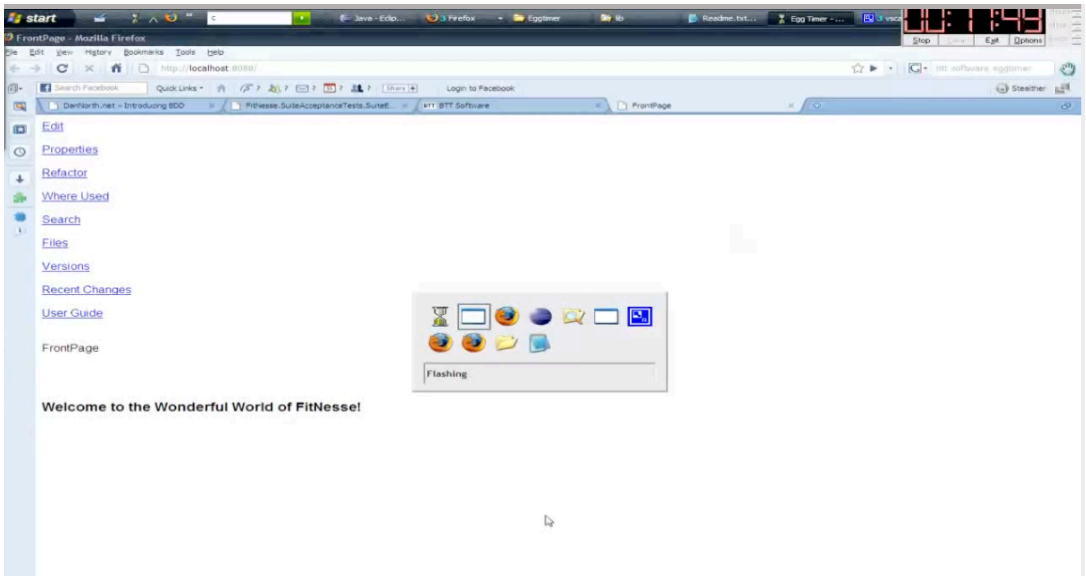


Figure 8.3: Something doesn't look right.

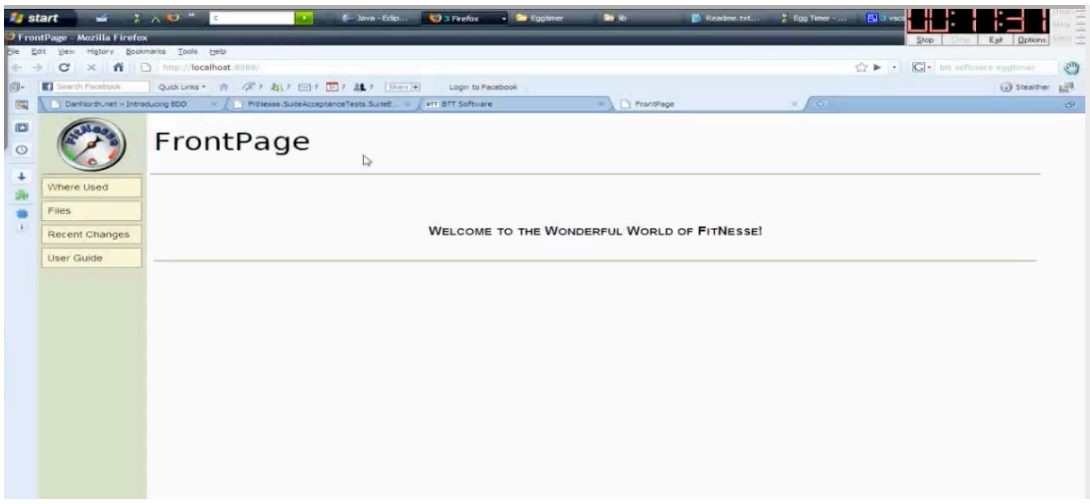


Figure 8.4: Now it is "hunky dory" fine.

8.1.1.2 Intrinsic information

Erroneous outcomes are not always evident: an action or words may seem reasonable to observers (Norman, 1981). These errors must instead be detected through assessment of intrinsic information. In the context of interactions with computers, actions may be taken in a software system that are correct, but which were not intended. The system has no information about the goals of the user in this case, and so cannot provide a response that

will trigger error detection. These errors must be assessed in terms of the underlying goals that directed action (Zapf et al., 1994)).

Conceptual integrity is believed to be at the root of many errors in software development. Because ideas are faulty, the logic goes, there are bugs (Brooks, 1995, p. 14). Imprecise requirements and poor design are often reported after the fact as a cause for faults (Basili & Perricone, 1984; Perry & Stieg, 1993). Across the sets of data examined here, developers frequently came up against barriers in their work related to intent. These encounters were often detected through ***explanation*** and identified in analysis by indicators of ***satisfaction and suspicion.***

This point is made strongly in Chapter 5, in Kasia and Bill's discussion of rather cars or intersections should manage traffic. We, as analysts, know that an error has been encountered by the use of questioning, fluid terminology, and also by the lingering suspicion conveyed by Bill at the end of the session that it is intersections and not cars that should have control over managing traffic. In this case, the error is one in forming intention, in determining how the software that the designers have been tasked with designing ***should behave***.

Marcus provided another example of detection through explanation. In his case, the problem was recognised when he explained an implementation to a person who was not directly involved in the project:

> ***Marcus****: ... So we want to make sure that [ClassName] was interacted with in a particular way, and in this case, umm, this is why it just doesn't feel right that this is, it's just too specific for it to be a [ClassType], dude.*

The utility of explanation in error handling has been observed before within software engineering (Knuth, 1989). It is a recognised problem solving tactic in software engineering trade discourse and commentary, variously described as the cardboard cut-out dog

(Baker, n.d.), rubber duck debugging ('Rubber duck debugging', n.d.)[5] and the teddy bear principle (Pascal, n.d.). Robert described the teddy bear principle like this:

> *"[You] put a teddy bear on top of your monitor and talk through your problem to that, and it is kind of that talking through the problem will kind of, kind of usually trigger in your own mind what actually you've forgotten to do or haven't done or something like that."*

The phenomenon has also been recognised beyond software engineering. Within safety science, it is the provision of "fresh eyes" on a situation that can allow "higher-level", knowledge-based errors to be detected (Woods, Johannesen, Cook, & Sarter, 1994). Explanation is a valuable tactic, but it does not always work. Robert related that he tried first to solve his problem by talking things out with another developer. He expected it to help, but in the end, he did not have enough time to achieve insight using the technique.

The account given by Thomas provides evidence of explanation that spurs detection. In Thomas's case an error was detected because he needed to explain the task to a colleague. When asked why parameters on pages should be named in the way Thomas was describing, the explanation he could give felt ***unsatisfactory***. All he was able to say was "that's the way the other screens have been done".

The full account suggests that it was in explaining the task that he lost intention. Thomas indicated in his interview that he perceived the issue not to be with his memory, but rather to be one of knowledge. He did not ***understand the naming strategy.*** He described the issue as one of "just getting the concept clear in the head". Frese and Zapf (1994) describe such errors as ***thought errors***, which may be due to forgotten intentions, but may also arise when a person lacks knowledge.

5. This wikipedia article (at date of access) gave the best overview to the concept, with links to related pages and printed sources. A copy can be supplied if it has substantially changed since access.

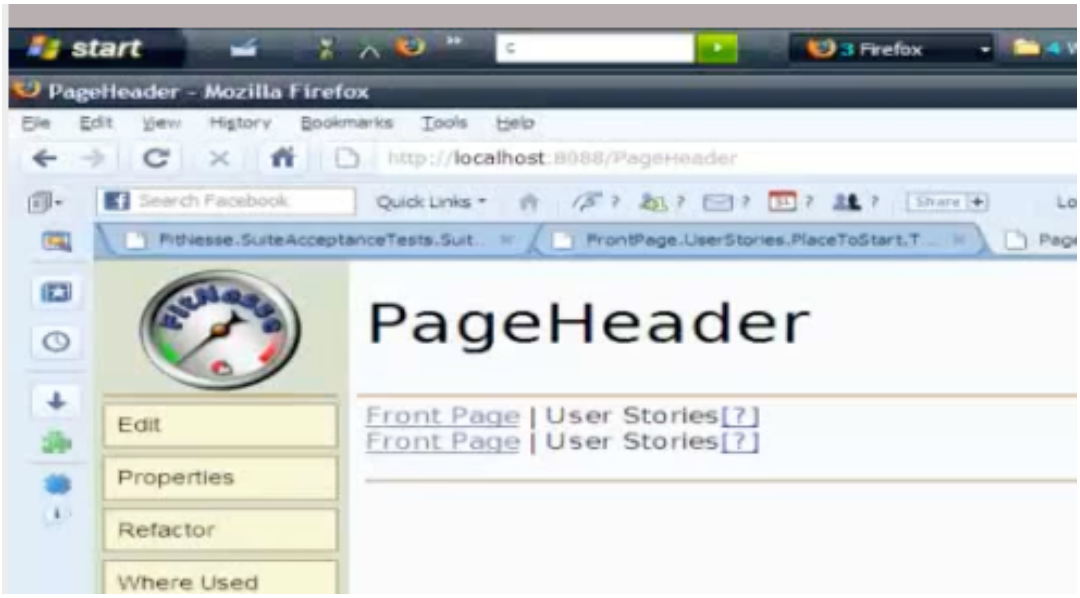


Figure 8.5: Errors aren't always evident. The wiki syntax entered to link to "User Stories" resulted in a to-be-created marker when rendered on the HTML page. This is indicated in the figure with a question mark.

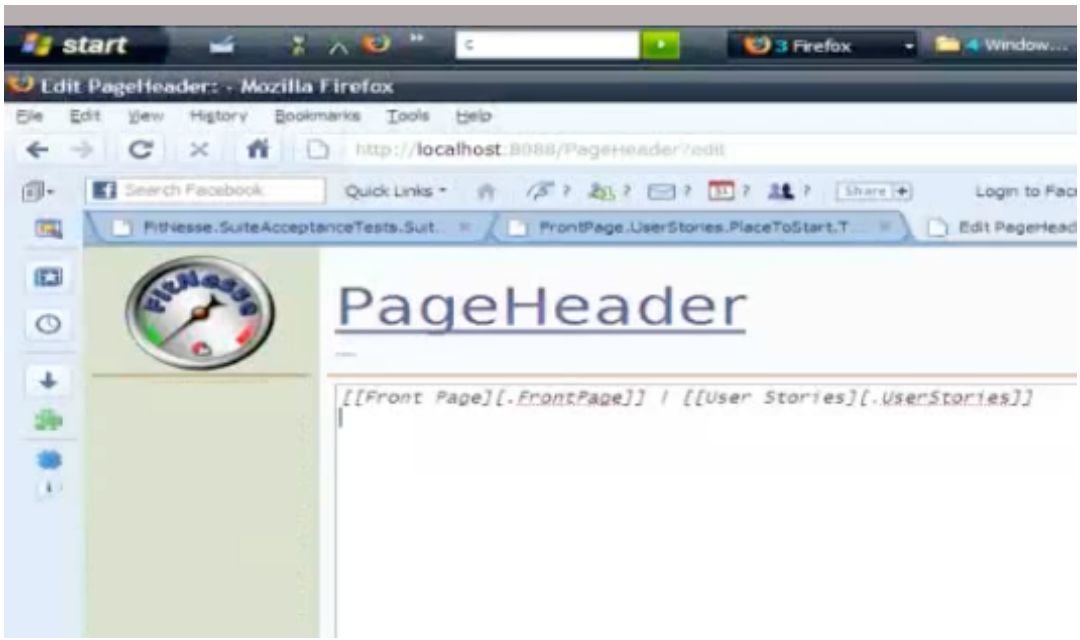


Figure 8.6: It looks okay to me.

Intrinsic information is often required to detect errors that have a conceptual basis, however such errors can be sneaky. They can manifest after simple actions that seem to have reasonable outcomes when assessed by one person, but be obviously wrong to another. Marcus and Joe provided an example of this in an incident in which Marcus tried to make a link to a different web page using wiki syntax. After saving the edits, the information on the web page suggested that a new page still needed to be created (see

Figure 8.5). . The syntax “looked okay” to Joe. His observation was sound, the syntax was correct (see Figure 8.6), however Marcus remembered that the data was wrong, because it did not reflect the information architecture of the wiki.

Errors of intent thread through practice. They arise during design activities, and come up again and again in the midst of development. They have been described in other studies of software development at the desk as being “like design”. In order to do programming, developers rely on "under-determined" matter: opinions, discoveries and alternatives that emerge "when-and-through" the practice of programming (Kristofferson, 2006, p. 10).

An example of this can be seen in one incident at the desk (issue 13-B in the catalogue located in Appendix C.2). In this incident, Marcus and Joe run into problems refactoring a method that manipulates two classes in the domain model they have created. The two classes are conceptually similar, and problems with them have come up before (see related entries 4-F, 6-A and 8-C in Appendix C.2). The error cuts across layers: Marcus and Joe are not sure anymore about how they defined the classes, or how to distinguish the classes from one another in this circumstance. Furthermore, they do not understand an implementation they made earlier that references the classes. The implementation uses a JAVA language feature with which they have limited experience.

Thought errors stick around, as the resolution of this issue demonstrates. Rather than push through to a fix, Marcus and Joe decided at a certain point to abandon the refactoring altogether, and to revert to the previous, working state of the code. The accounts of tolerating and settling demonstrate that such errors can remain active for long periods of time, managed by making incremental progress through verbal consensus or temporary solutions that satisfy the developers and permit the issue to be set aside. This does not mean the issue is resolved. In most cases, subsequent instances will occur in which handling must continue.

8.1.2 Identification: Knowing what should have been done

Once an error has been detected, a person must identify what was done wrong, and determine what should have been done The examples given action-based studies of error suggest that insight is unambiguous, that identification and recovery are more or less instantaneous once a detection has been made. Across these sets of data, error handling is often simple and compressed, but other patterns were also observed.

When an error is detected, developers do not always know how big the problem will be, or what kind of problem solving will be required. This understanding comes through identification. At the desk, programmers must proceed in all tasks by first establishing the "fact" of what they are looking at (Kristofferson, 2006). This is also true in the case of errors. Identification is not stepwise or linear. Developers do not consistently recognise that a problem exists, then diagnose why the problem happened, then implement a mechanism to fix it. Evidence is given that they search for commonalities between prior experience and the current situation (Rasmussen, 1985) within a cyclical process. Identification often requires ***multiple rounds of local problem solving.*** This concept is depicted in Figure 8.7 below. Guided by system responses, information gathering (Eisenstadt, 1997) is interspersed by manipulations of the environment.

If one cycle of problem solving fails, developers must deal with newly created changes in behaviour, as well as considering the previous conditions. This requires them to keep track of what they have done, what they have tried, what they have changed. The risk is obvious: if they do not work forward slowly and systematically, they will forget what they have tried and the order in which they have tried things. The need to manage and remember state is vital. Robert described it like this in the context of web development:

> *"Sometimes you're making changes and then you'll try to reproduce the issue and go 'Oh well that had no effect' but you have to be sure that the change you've made has actually been picked up, because sometimes when you are dealing with web*

things, something might have been cached … even though you've changed something, maybe you forgot to save the file, maybe your change hasn't been picked up…"

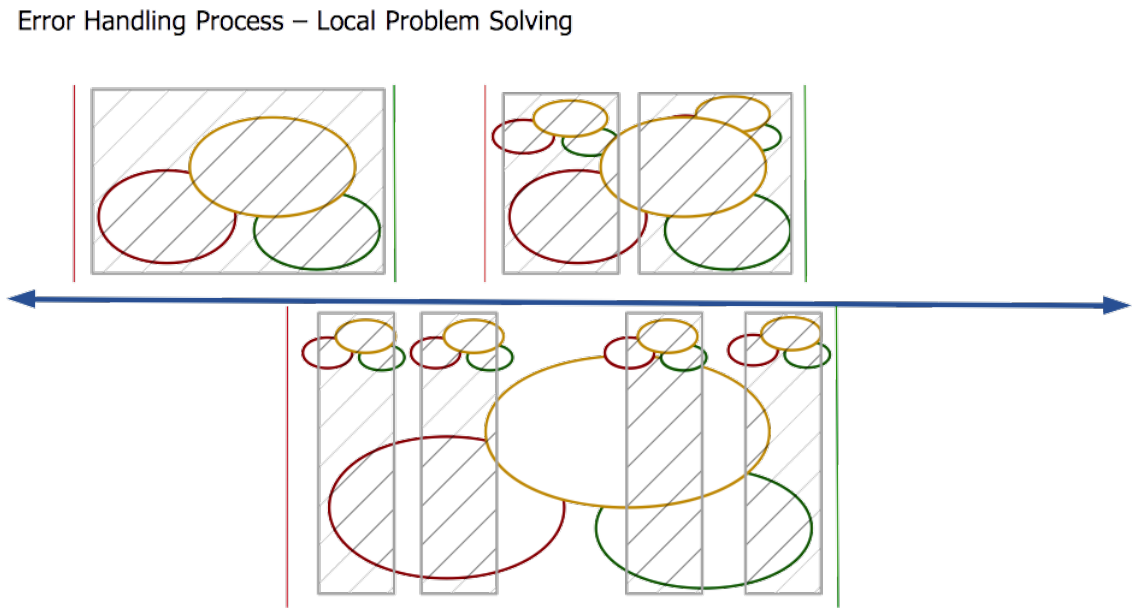


Figure 8.7 Error Handling Process - Local Problem Solving. This diagram presents a normalised depiction of local problem solving. The example on the top left replicates the basic process depicted in figure 8.2. The other two examples use grey hatches to indicate individual instances in which local, immediate goals and actions are undertaken, observed and assessed. This diagram doesn't represent scale, only that some errors can be solved in a single instance of handling, while other problems require multiple tries at different points in time, indicated with the blue arrow. Each of the hatched instances is depicted with an accompanying small error handling process; these should be associated as belonging to the overarching process represented by the large set of bubbles, and by the delineation within red and green halt and resume progress bars.

Assessing state is a tactile and immersive experience. Evan provided a clear example of what this feels like in his working environment:

"I work on a small desktop Mac, so I have a Mac connection to the computer at work that is running the virtual machine and so I was getting rather annoyed trying to navigate between three different screens on one 13-inch monitor, and getting rather confused in the process about what I had done and what I hadn't done and what I'd changed and what hadn't changed."

The time it takes to identify sources of errors varies enormously. Sometimes identification takes only minutes, but it can take hours, days or even months. The length of time

required relates to the information that is needed, and what the developers do with it. Developers develop understanding over time, as their frame of reference shifts in response to new information (Rizzo, Ferrante, & Bagnara, 1995). At the desk, Kasia and Joe were able to control identification by setting constraints on responsibility. Marcus and Joe needed to identify factors in the immediate environment to return software to a working state. Developers reported after the fact that identification sometimes is drawn out to develop technical understanding about domain requirements and client priorities.

Extended timeframes are useful, they can give developers time to think and to explore possible solutions. Valentin described how he tolerated multiple manifestations of an error for over a year and used partial, temporary solutions along the way to gain time to consider a proper solution. He reported the decision to postpone as strategic, a behaviour that is consistent with descriptions of expert debuggers in other studies (Freeman & Black, 1992).

On the other hand, individual cycles of local problem solving are described using terms that confront notions of expertise as strategic or principled. Developers explain what they do as "hacking around", "trying things" "trial-and-error", "attrition", or "nested problems" Valentin described the experience in this way:

> *"You find something, and then you find something else related, you find something related and you are deep in a tree where you [are] never at the end and you must come back."*

Insight about what was done wrong and what should have been done often must be ***pieced together.*** Robert described finding his solution using a painstaking approach of analysing information taken from the internet to identify things to try in his source code. He reported that the technique was not a strategic "working down", but rather was driven by trial and error, by trying things out. He assessed it this way:

> *"[I]t was obviously a case that there was a solution to this problem, it was a case of working out what I was doing that was different or what other people had done*

> *that was different to what I was doing...it was kind of understanding the solutions that they had posted because they would only post the parts that they felt were relevant."*

Developers come up against boundaries to action set by the tools they use, the code they call on, and the social environments in which they work. These are commonly described in safety science as constraints on the space of opportunity in which they work (Rasmussen, 1990). Within psychology, errors at these points are often described as latent, cases in which decisions taken at a different point by other people have disastrous effects (Reason, 1990). Within software engineering, boundaries are commonly conceived of as interfaces, points at which developers must utilise software defined by others.

Boundaries of all kinds, including interfaces, test belief. Interfaces are known as features of software architecture but they have also been found to be social: bridges between teams, departments, and the world "out there" (de Souza, Redmiles, Cheng, Millen, & Patterson, 2004). Development practice depends on programmers' belief that code written by someone else is correct (Naur & Randell, 1969). However, interfaces have been shown to have a high incidence of faults (Basili and Perricone, 1984; Perry and Evangelist, 1985, 1987). They are hard to learn (Robillard, 2009) and they both facilitate collaboration and isolate developers (de Souza et al., 2004).

Interfaces lie at the boundaries of responsibility, in decisions taken elsewhere. They can have effects that are perceived as errors by developers: something is not right, and must be handled, but symptoms and factors are opaque. In these cases, it is often impossible for a developer to determine what was done wrong. And it is likely also unimportant for a developer to understand why something is wrong. The developer is not responsible for erroneous behaviour they encounter in these cases; they just have to deal with it.

8.1.3 Recovery: Removing effects

Recovery does not follow heroic bursts of creative, intuitive performance (Cross, 2001). Insight is often described by developers as being sudden or serendipitous, but this is not supported in the evidence. Instead, insight, and by extension recovery, is more often achieved through outcomes of problem solving that are ***perceived to be timely***: the combination at some point of accrued knowledge, memory, recognition, and evaluation.

Recovery should not be equated with resolution. Issues may remain active because recovery is impossible, as in Dereck's case, or because details of a local occurrence actually belong to a longer thread of practice, as in Joachim's case. Sometimes a recovery will have consequences that contribute to or shape other priorities. These might be individual ("I can see know I need to write cleaner code"), team-based ("We are just going to re-architect into a single solution") or with an eye toward the collective needs of a development community ("I thought there ought to be a way for others to use this code").

The first and foremost priority of software development is to keep work moving. Keeping work moving does not require that a developer understand all of the details about why an error occurred or what removed effects. Things are often left unknown after recovery. Developers may achieve a working solution, but may not be completely sure which of the steps or the order of the steps that fixed the problem.

Upon reflection, both Joachim and Robert are able to name errors in understanding about how the libraries they were working with behaved. Both left their experiences with gaps in understanding. Robert described more than once his expectations for how a library behaved versus how it actually did behave, and commented that he still didn't understand why it behaved as it does. In Joachim's case, lack of understanding was expressed in terms of doubt in an API that led to a change in technology after we spoke.

Barriers to recovery can come out of policy decisions taken long ago. In Dereck's case, the policy to delete old backups after a certain time limited options for recovery. Other decisions to limit administrative access to the release tool blocked recovery. Dereck had only a vague understanding of the history of these policies and the effects they had on the software he was trying to use. They produced barriers to his recovery, but they were not exactly errors.

Knowledge gained in the course of fixing is often situated and circumstantial. A fix might arise, for example, through an assessment of syntax ("Ah, it needs to be on the other side."), without evidence that the developer has gathered a profound understanding of the language feature itself. A developer can learn during handling about the requirements and expectations of particular software frameworks, but the general knowledge, for example that some packages will require files to be located in specific places may be of limited utility going forward. This event, however minor or severe, will likely never occur again. Circumstances next time will be different, the locations will be different, the cues that give rise to the issue will be different. Developers give the impression that insights drawn from one experience are not sharp, crystalline pieces of knowledge that can be plugged into new problems.

That said, developers do appear to make something of handling experiences. With only a few days separation from the event, developers are able to articulate an awareness of severity during handling incidents. They are able to identify when problem solving was ineffective and are aware when they got lost. They are also able to identify when limits in their own understanding contributed to the problem, and display awareness of problems that could occur again.

Interestingly, even severe incidents were reported as having positive outcomes, but this may be the result of human impulses to make the best of "personal failures". Developers

reported that they had achieved a greater level of comfort and understanding of the software frameworks they were using. Getting stuck forced them to examine aspects of the software that they had previously taken for granted. They had to learn more about the software, quickly, in order to solve the problem and resume forward progress.

8.2 The Shape of Experience

Errors are sometimes reported, and error handling will likely begin with ***replication*** and ***witnessing,*** themes that were discussed in more detail in Chapter 7, Section 7.4.2. In cases in which an error comes up in the midst of work, it is ***experienced*** first-hand, and information is given by the system or by internal perceptions that something is not right. Handling is often impulsive at first, marked by doubt, claims of innocence and blame but settles into investigation of behaviour that has been observed or reported.

Experience shapes error handling processes and handling, in turn, forges experience. The process takes an individual shape formed by expectation and other feelings, by getting things wrong, thinking of similar experiences, and seeking support, as depicted in Figure 8.8 below. As the paired work analysed in Chapters 5 and 6 demonstrate, errors that arise in the midst of work are often conveyed with surprise, and handling is punctuated by feeling: with questions, expressions of doubt, and by placing blame. After the fact, developers take the blame for decisions they deem to have been badly made through indications of dissatisfaction, by naming what the problem was, or by expressing lingering doubts.

8.2.1 Expectation and Surprise

Errors surprise developers, the conditions in which they arise, and the behaviours they produce are unexpected. In many cases, developers are at least momentarily stumped while they try to identify the source of a problem. Expectation and surprise are known to

be components of error occurrence. Ko and Myers noted that their programmers asked questions when something failed in relation to prior formed expectations (2005).

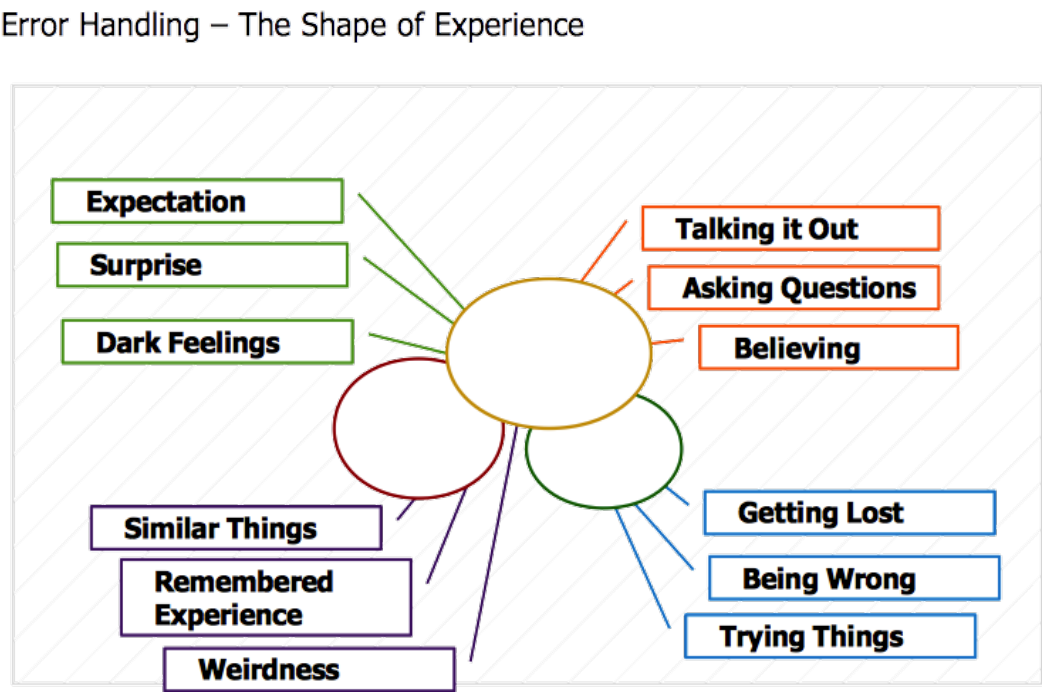


Figure 8.8: The Shape of Error Handling Experience. The error handling process is represented by the triad of coloured bubbles established in figure 8.1. In this instance, the triad could represent an entire incident, or one instance of local problem solving. ***Feelers*** in different colours depict modulators that focus attention, redirect activity or develop perspective.

Expectations are closely linked with suspicion. Suspicion is a feeling, a sense that something is wrong or has been done wrong. Suspicion does not guarantee that an error exists, and error handling is sometimes undertaken in the absence of errors (Allwood, 1984). Developers often think something in the code is wrong that turns out to be correct, a process that has been described within software engineering research in terms of hypothesis formation and modification (Lawrance et al., 2013). Errors are thus sometimes expected ("I expected it to happen."), but expectations can be defied:

> ***Joe***: *And we expect this to fail. It is going to say, I can't find a dummy role with a small 'D'. Oh- and it has, look, it has got a little red thingy saying NoClassDefFoundError. That's funny, I thought it was supposed to say, uh shouldn't that come back as a casting? (Episode 7, 11:13)*

During identification and recovery, developers try things that they believe should work based on their understanding of the environment, knowledge of languages or past experience. Sometimes, they are not rewarded with an outcome that matches their expectations. They are surprised again, confronted with new behaviours that must be assessed. Getting back to the original error state can be difficult, and in some cases is impossible. They can get lost.

Recovery brings with it fresh expectations, and so it can be said that surprise and expectation permeate all stages of handling. Evan expects that he will have similar kinds of problems when he promotes his software to a different environment. Valentin reported that he expected to see occurrences of the rendering error based on past occurrences. As he described it, "this [the first occurrence] prepared me for that".

8.2.2 Feelings

Findings suggest that ***dark feelings*** are used by developers much like ***bad ideas*** (Dix et al., 2006). Dark feelings can be used to expand and constrain the problem space, allow developers to consider unlikely sources of error and to direct investigative activity. If developers suspect a problem, they commonly doubt or question an outcome ("Why is this happening?") or place blame ("What have you done?"). When they cannot form intentions, they might indicate that they feel stuck, that they are at "an impasse", that they have "no ideas", or that they are "poking around in the dark". Error handling can make developers feel bad, they can get "down on themselves", or become frustrated. They wonder why they can't figure something out or aren't performing well.

During particularly effortful problem solving, handling has been shown to be stressful and uncomfortable (Brodbeck, Zapf, Prümper, & Frese, 1993). Keeping emotions under control has been shown to have an effect on learning new tasks (Keith & Frese, 2005). Findings indicate that stress is marked in developers by verbal expressions, and by what

they do. Troubleshooting efforts move from examining inputs to an unexpected condition, to the way concepts are represented within source code, and even to elements of the conceptual design itself. They are prone to doubt early ideas about the sources of errors, and the means to fix them.

It is clear from the accounts that were examined that feelings also inform decision making in broader terms. Developers describe rejecting alternative solutions on the basis that that they are "very ugly," something to be used only as a "last resort". ***Ugly feelings*** linger even after recovery. Evan got his application framework to run, but is aware that it is "pretty dirty".

Developers leave encounters with a sense of how well solutions are functioning that at times contradicts cues given by the software or other artefacts. Their sense in these cases is often one of suspicion, of caution. Joachim achieved a working solution in his design, but was dissatisfied, he felt something was not right. Evan had no information after recovery that things were not working, but was still wary. As he put it, everything is working now "touch wood".

The findings clearly suggest that handling processes are modulated by emotion but it is not clear if emotions serve or hinder the process. Negative emotions have been found to produce negative effects on task performance during software development (Graziotin, Wang, & Abrahamsson, 2015), a view in line with the evidence given in this thesis of thrashing, turbulence and severity. Questions on these point remain, however, and should be addressed in future research. Does it matter if developers feel bad during or after an error handling experience. Do dark feelings change how they behave in the future?

8.2.3 Similar Things

Developers draw on similar experiences to assist error handling. Analogies are drawn from past work experience, projects that may have had similar requirements, or from prior

experience adapting or administering software. Like experiences or knowledge may be shared with a developer by a colleague in conversation, email or other shared written sources. They equally might be formed through more general experience in the world or with software as a user of environments that are similar to those being built (McDonnell, 2012), for example of simulation software or web browsers.

Analogies may provide information to support an error handling process, but are also used more generally as a part of decision making for tasks. In the former case analogies are accessed during brief moments of reflection in the midst of problem solving. In the latter case, they are a part of deliberative activity (Eraut, 1994), such as the time Kasia and Bill spend considering how traffic works in the world, or in reports by Valentin about alternative solutions he considered for rendering special characters.

Analogies are called to mind, they are remembered. As might be expected, memory is faulty, and recollections may be partial, serving as impressions that cue detection or delay resolution. Developers may remember having done something or having seen something, but may not be able to remember what they did exactly or where they saw the information of interest.

Though it would be difficult to claim that analogy leads developers astray, there is some evidence that analogy can increase stress during handling, particularly if an analogy drawn from prior experience is perceived to be similar to the current situation, but outcomes are somehow different this time. Turning to existing code, for example, is a useful tactic during handling. It may offer a template for a fix that does not need to be proven.

However, depending on code to provide information for fixes may also hinder development of understanding. Marcus and Joe relied on a prior implementation of JAVA generics to piece together a solution (see also Appendix C.3.5), but understanding of what they did was limited to material aspects of the syntax that would get the software to run.

Fixes are not made by accessing or relying on similar experience. Analogies must always be compared with other information, they must be assessed and shaped to match the requirements of current circumstances.

8.2.4 Seeking Help

Errors are individually experienced. Error encounters can demand or encourage social support, but this does not always come in the forms we have come to expect by studies that advocate for "soft skills" to improve dependability. Information is drawn out of tips or stories passed along by co-workers in conversation. Developers also make use of open-source documentation, and have access to subscription-based sources that provide industry-tailored trade films and texts. *Ad hoc* (Ko et al., 2007) and formal teamwork (Seaman and Basili, 1997) are helpful in some cases, but social support increasingly comes from other sources found on the internet.

Wiki-posts, internet fora and websites are widely used to identify technologies, to assess reliability of open-source products and for more detailed information about how to solve particular problems. Robert explained that depending where you are "on the cutting edge of things" it is normally possible to Google an answer for something. Usually questions have been asked in places like Stack Overflow and there are answers that can be studied. Several other developers provided information about other on-line trade publications that they used, and explained how they hone search terms to find solutions for similar problems.

Help is not always easy to find. Developers may dutifully seek out colleagues and even prefer this practice over the use of documentation or internet sources. However, they may not always be able to get time with or find colleagues with skills or experience that are useful for the problem at hand. Fora are described as being intimidating, or high-minded. Formal documentation does not always yield information, and documentation provided by

open-source software sometimes hasn't been written or is incomplete. In spite of this, social support in all forms is valued: developers seek each other out when they can and report that information from the internet, however partial and scattered, makes work easier and smoother.

Social support for handling is sought from a collective formed of colleagues, from in-house documentation produced by development teams, and from commercial and social sites on the internet. Information gathered must be transformed into guidance. This is done by searching for corroborative testimony, by writing toy implementations and making tests. Only then will it be deemed by the developer to be both ***useful*** and ***trustworthy***. Often, within a handling process, this is when the hard part begins. As in the case of analogies, the information must still be fitted and matched to the problem, the developer must figure out the "special part" that will make the guidance work.

8.2.5 Weirdness

Just as detection can be made through developers' sense of things that "look wrong", handling can be hindered by ***weird behaviour*** in tools. Developers are accustomed to bumping up against constraints in the tools they use, it happens all the time. They may not understand how a tool works, or how to access features. One tactic often taken in these cases is to ***try another way.*** The developers do not question the behaviour of the tool, but instead take swift decision to accept the constraints. They also may not understand afterward why "the other way" worked.

Developers accommodate weirdness in tools when they can. This is practical, because weirdnesses related to state can mysteriously resolve and it is not always wise to divert practice to address responses given by tools. Errant behaviour in tools can be more severe, however, distracting from or hampering progress. In extreme cases, it can overtake development, and require developers to give over problem solving to evaluation of

unrelated problems. These experiences can lead to distrust, triggering suspicion in future circumstances that errors exist even where they don't. Even in smaller cases, weirdnesses can persist, and at a certain point developers may have to accept that they do not understand what has gone wrong, and recover by removing or downgrading installations.

Spurious errors that come up at boundaries are leveraged by developers to learn about parts of a system that are infrequently visited or aspects of a technology that are not well understood. They encourage or demand that developers engage at a deeper level with third-party software libraries or code written by someone else. This was particularly true in cases in which the thing that went wrong resulted in problem solving was turbulent. Though the "other" software formed a boundary that prohibited work in the short term, in most cases developers recognised the boundary as an invitation to learn.

8.2.6 Being Wrong and Getting Lost

Developers get things wrong all the time. They make mistakes during conscious, laboured reasoning that characterises higher-order problem solving, but also while undertaking small material actions within a language or environments. Being wrong is an effect of guessing and of trying things. It may come out of a half-hearted proposal that is made and followed ("It might have to go after the dot. No."), but may also arise out of identifications made with confidence ("Ah ha ha! I know what it is!").

Developers don't always understand problems, technologies or have the necessary skills when errors occur. They recognise this as a central factor of many handling processes they engage in. Specific training or knowledge gained beforehand might prevent problems from occurring, but developers dismiss this possibility when asked what they might have done differently. They explain instead that it is more important to be able to ***gain understanding*** when it is needed. As Robert explained:

> *"If I was going to go back and approach the issue again, I would try to make sure that I did understand what was going on in the framework upfront, but there is so much to know that you just need to make sure that you understand enough to make it work at this point in time."*

Developers are comfortable with being wrong. In a period of reflection, Marcus remarked, half in jest, that being wrong is the "whole point" of development, noting:

> *"[Y]ou spend most of your time when you're developing stuff being a little bit less wrong than you were a few minutes ago so. So we're always wrong, technically speaking." (Ep. 19)*

Comfort with being wrong and accepting limits to understanding may be outcomes of the demands of practice. Decisions are sometimes hastily taken, code slapped down, perhaps due to pressures on time, but also because it is pragmatic to work this way. Sometimes being wrong is strategic, it is employed. Bugs are allowed to reoccur, giving developers time to do other work, but also to learn. By observing behaviour in software and the effects of behaviour on clients, developers develop understanding about priorities and technologies, and identify what they still need to work out.

In paired interactions, developers do not appear to penalise each other for getting it wrong, even when a mistaken idea results in code that takes a significant time to implement or which has to be reverted. Likewise, developers do not always express shame at having done something wrong, even if the error results in a bug that is public-facing. They will make the same mistake again and again if it supports their preferred practice. ***Some errors don't matter,*** particularly when considered in terms of the priorities that a developer has.

Being wrong may be a matter of course in developers' lives, but sometimes, the simplest errors can turn out to be the most ***severe***. In this case handling takes a large amount of (relative) time, demands multiple, intense rounds of local problem solving or causes stress or anxiety.

Severe incidents begin in much the same way that other error occurrences do: an error condition arises that is unexpected. A developer begins a process of investigation, making guesses about which of his previous actions and decisions resulted in the problem. He gathers information, perhaps by examining areas of the code that may be related, and trying things out. The difference between severe instances and other incidents is that efforts do not yield information or changes in program state that remove the error. Furthermore, the incremental outcomes do not "make sense". Things can take a turn for the worse and go horribly wrong.

Simple issues that turn out to be severe are surprising: one might not expect experienced developers to get stumped by a configuration problem or by a class path issue, but self-proclaimed novices and experienced developers engage in similar handling processes while solving these kinds of issues. The suggestion given in this data supports findings in problem solving research more generally (Reason, 1990) that novices and experts ***get lost***, and when they do, they exhibit the same ineffective behaviours.

Within the catalogue, two other incidents that might be characterised as the most severe occurred during simple, routine activities like the one that tripped Dereck up. Evan had formed a sense in prior work of the tasks involved in getting a software package up and running, but spent a considerable amount of time fiddling with configurations in the wrong file trying to get things to work. Marcus and Joe likewise had an issue with configuration and knew they needed to check to see if a file was in a directory. What they failed to notice, however, was that they checked in the wrong place.

Severe incidents may be critical if they have effects beyond the developer's individual experience. Dereck knew that he needed to manually copy files to the server, but committed a slip in execution. His issue was ***critical***, because the error resulted in his team breaking a contract of service in the department. The handling was also severe because his

ability to act was constrained by circumstances in the environment. The issues had several simple resolutions, but handling developed into a stressful and uncomfortable experience. Dereck knew what he needed to do, but was not able to perform necessary steps for recovery.

8.3 Limitations

It is difficult when using naturalistic observation to determine what should "count" as an error (Norman, 1981, p.13). Analysts are not usually able to establish causes based on observation alone (Hollnagel, 1998, p. 78). It is not always practical to determine whether or not an informant had a wrong intention, the criterion by which errors are generally categorised as having been mistakes (Reason, 1990). Likewise, the information available in research data may not allow an analyst to determine how well or by what means an informant "understood" a situation they were in.

The research in this thesis has not established causes, but has noted behavioural aspects of error occurrences, that is, what was done when the error occurred, such as omissions, insertions, substitutions and reversals (Reason, 1984, p. 530). Naturalistic data about error is by its nature selective, and so the studies reported here may have descriptive power, but cannot be put to predictive uses (Norman, 1981).

8.3.1 The Vagaries of Access

With few exceptions (Prior, 2011), empirical studies in software engineering must make use of opportunistic, short-term access to field sites. Access is often constrained, and management can place severe restrictions on research design and reporting (Perry & Stieg, 1993). It may only be possible to observe developers at the desk for short periods of time, researchers may have to rely on mixed collections method including "serendipitous observation" (Robinson et al., 2007, p. 541). More commonly, in-depth knowledge of developer practices is reported by industrial researchers with longstanding experience in a

company (Endres, 1975), or emerges from a series of studies undertaken by the same group of researchers over time (Aranda & Venolia, 2009; Guo, Zimmermann, Nagappan, & Murphy, 2011; Ko, DeLine & Venolia, 2007).

Limitations to access were not overcome for this research, they were worked around. Relatively unstructured, open access was gained to sites at which to conduct interviews through contacts within standing professional and academic networks. Collection was tempered with gleaning, by seeking data from within sources that had been collected by other researchers and professionals.

8.3.2 Credibility and Reliability

One way to improve credibility in qualitative research is to have more than one researcher collect and interpret data, a solution that was not possible for these studies. Instead, concurrent triangulation (Easterbrook et al., 2008), was undertaken by gathering and comparing data from multiple sources that represent different aspects of development work. Data was compared for points of similarity and difference. One source of data was used that is publicly available, and the methodology used in analysis has been documented so that other researchers can assess its credibility (Robinson et al., 2007).

To supplement information lacking in one set of data, evidence drawn from different studies has been used not only to triangulate, but to build up contextual understanding, a technique that has been described as colligation (Anderson, 1997). This was necessary because the nature of the data sources reflected different kinds of problems and different tasks and were recounted in varying degrees of precision.

Fieldwork is said to be less reliable than other data collection methods, because collection is so personal, a weakness that can be exacerbated when the researcher is close to the environment studied (Robinson et al., 2007). Ethnographers ultimately must overcome limitations of closeness through the development of their reflexive sense, the

way they come to consider both the insider and outsider perspectives (Hammersley & Atkinson, 2007). Straddling these positions permits researchers to "'know' in ways that others don't and can't." (Anderson, 1997), however the limitations associated with closeness will always remain. Researchers are a part of the world they study, and data can never be "pure" (Hammersley & Atkinson, 2007).

8.3.3 Fixed Records

Chapter 3 argued that retrospective analysis cannot provide a full explanation for error in software development, yet the corpus was formed largely from secondary sources (McGinn, 2008), or concern work that was performed in the past. There are clear limits and risks associated with this approach.

When given indirect access to materials, researchers must infer behaviour and action that was undertaken in the past, to study both in terms of "material traces" that are "fixed" (Scott, 1990, p. 4). In fact, analysis was undertaken without any direct access to the situations described in accounts, and in some cases with limited access to the informants or creators of the materials.

To counter these limitations, a considerable amount of time was spent assessing the gathered material for quality, completeness and representativeness. The video recordings are records of work that were created with an eye toward the public (Scott, 1990), and analysis has accounted for ways in which the material is more reactive (Laurier & Philo, 2006), and in some cases not as complete as might have been hoped. It has been necessary to tease out the ways in which the accounts and exchanges were "geared" both toward the immediate, practical needs of the people depicted in them, but also toward the needs of the collectors (Scott, 1990).

The perceived and documented weaknesses associated with reactivity (Laurier & Philo, 2006) aside, the videos used depict a fair amount of unstructured, naturalistic exchange.

Participants who create video provide their own record of how they "view their world" (Hammersley & Atkinson, 2007, p. 149) and this has been leveraged in this research by examining speech for reminders that sources may be naturalistic, but are not natural.

8.4 A Partial View

Human error has long been understood and explained with examples that illustrate the characteristics of error (Norman, 1981). In a similar manner, the analysis in this chapter was intended to give prominence to developers' own voices in highlighting issues related to error. The aim in this and the prior three chapters was to convey how error is handled during software development by establishing a set of accounts that describe ***what happens***, and what developers **make of it**.

By using this perspective to examine sets of data that depict development in different contexts, an understanding of error has emerged that better reflects programming as a human activity (Capretz, 2014). While it is hoped that the effort has been progressive, that the heights are higher and the skies clearer than they were before, it is recognised that the view remains partial (Horst, 2009). This is only a start.

9. Conclusion

The developers who informed this research would probably agree that most errors in software development are due to problems of understanding, or of mismatched skill. They would question the corresponding views on expertise that are conveyed by these terms. Expertise in software development is not something once achieved that never changes. Development practice is marked by time (Winograd & Flores, 1987) and influenced by the larger environment in which it takes place (Curtis, Krasner & Iscoe, 1988). The knowledge required to develop software cannot help but change.

The tasks performed by developers are likewise active, continuous and dynamic. Performance is underpinned by skill and ability that develop over time. The problems encountered by developers are often novel, they require new knowledge or skills to be employed, or represent activities that the developers do not routinely perform. The problems are "new to me".

Though determination of "fit" in software engineering is commonly recognised to be dependent on an agreement from a requestor that conditions have been satisfied, findings better support Rasmussen's view that assessment of what is appropriate is personal (1985). Requestors will state that a condition has been satisfied, will determine strategic success, but this can only be done after-the-fact, with the value of hindsight and once the perceived goals have been established and achieved (Reason, 1990).

At the desk, at the drawing board, it is the developer who must assess and reassess "fit" according to their understanding of what is needed *right now* to keep work moving. When errors occur, developers need to understand what they are seeing, they need to be able to do things at this point in time with their tools, within their capabilities, and in light of their personal, project and organisational goals. They do this, in part, through error handling.

Errors usually aren't left behind, they are put right before a file is released, committed, or saved. Developers use error occurrences to form or test expectations for how software should behave, to direct tasks and to verify higher-order task completion. Analysis of error provides one way to examine software developers that reflects their work as makers of "grand conceptual structures" (Brooks, 1995, p.7) but also as operators of tools and users of systems.

9.1 Implications

Identification and recovery hinge on the ways in which developers assess the current moment, but also on how they come to modify practice over time. Endres suggested that developers form their own theories about why things go wrong, and that as a result, they modify programming style. He described this as a learning process (1975), and it is here that the greatest implications for this research lie. Understanding more about error detection and recovery stand to illuminate how developers develop competence, how they learn and grow.

The importance of information gathering to problem solving is established (Lawrance et al., 2013), but what is done with the information, just how it is transferred from book to practice or from one experience to another (Eraut, 1994) to form understanding should be explored in more detail.

Greater attention needs to be given to the nuances of ***error-driven practice***. Reason notes that there are often more forcing functions presented to operators while taking something apart, in which each step in taking something apart is "is cued by the physical characteristics of the item." (Reason, 1990). In software development, by contrast, inabilities to compile, to run, continually stop the ***putting together***.

Developers are complicit in this, adopting methodologies like test driven development that require them to continually fail forward. It is clear that the errors presented by

systems are leveraged and relied upon, but more investigation can be made do discover how this is done and the purposes it serves.

Likewise, findings suggest that ***feelings*** inform and modulate decision making in the midst of error handling, but it is not entirely clear how or to what purpose. Emotion may serve or hinder developers, or it may be that feelings are of principal use to researchers, providing verbal signals or hooks into experiences that are under examination. More investigation should be made before drawing either conclusion.

9.2 A Framework for Examining Practice

To understand the active qualities of error in software development, relations rather than causes have been examined. Incidents were constructed out of fine-grained information provided by informants. Public-facing critical and personally severe incidents were examined, but so were everyday issues. Local problem solving was examined in development practice that is primarily strategic (Reason, 1990), forward looking or deliberative (Eraut, 1994) but also as it unfolded at the desk.

Analysis drew out decision points that relate to the error handling stages, and examined modulators of handling processes including blame, suspicion and doubt. Developers' temporal orientation toward software was examined: how they postulate about the future, describe things in the present moment, and reflect on recent experiences.

This approach advances empirical studies of software development in two ways. Theoretically, it expands the conceptual space for error in software engineering by providing insight into errors that occur between commits and releases. In so doing, error is permitted to be a normal aspect of development practice. By enlarging our understanding of the role errors play in software development, we are positioned to enrich our understanding of how competence, knowledge and skill develop in the circumstances and situations that comprise daily practice. If not grand or universal in its achievements

(Ekstedt, Johnson, & Jacobson, 2012), this research has described how errors are encountered and handled by professional developers. It joins other recent efforts (Päivärinta & Smolander, 2015), in establishing a framework to situate findings related to professional practice that are not tied to specific software engineering tasks, tools or methodologies.

9.3 The Changing Nature of Expertise

The timeframe in which incidents are examined and the perspective developers hold toward them are significant. Gathered early in a development process, accounts may lend themselves to categorisation as skill-based errors of action. It is easy to conclude that they were committed by novices or due to incompetence.

When accounts are gathered in the midst of work or are constructed out of evidence representing a longer arc of time, murky areas of practice emerge that require closer examination of contextual and circumstantial details. These details may need to be carefully tracked or elicited after the fact, because the practice with which they are intertwined may be so fresh or unformed that the developer (and analysts) may not have the benefit (or weakness) of hindsight.

When practice is examined over time and out of the bounds of tasks and software engineering methodology, we are given a sense of how expertise ***changes,*** of how it develops and grows. This sense is formed by observing practice, but also by listening to what developers say**.** Developers use future facing statements to create boundaries around the problem solving space, to constrain and restrict problems and to limit responsibility. When developers narrate the present moment, they are more affective, the actions they take are often tactical. When asked to reflect, developers are at once astute and unguarded.

Looking back, developers tell us how technologies work or about how the wheels of organisations grind. They may give indications that the knowledge they share was learned as a consequence of the issue they are discussing, but usually such a connection can only

be surmised. If we listen closely, developers reveal what they ***needed,*** what they ***did not know***, what they ***did not realise*** in the midst of an issue, or what they ***still don't understand***. They also tell us what they believe they ***should have*** done, they indicate practices they ***would like to*** routinely follow If we continue to listen, they will tell us even more.

References

Allwood, C. M. (1984). Error detection processes in statistical problem solving. *Cognitive Science*, *8*(4), 413–437.

Amalberti, R. (2001). The paradoxes of almost totally safe transportation systems. *Safety Science*, *37*, 109 – 126.

Ambler, S. (2012). Introduction to test driven development. Retrieved from http://www.agiledata.org/essays/tdd.html

Anderson, B. (1997). Work , ethnography and system design work, ethnography and design. In A. Kent & J. G. Williams (Eds.), (Vol. 20, pp. 159–183). Marcel Dekker.

Aranda, J., & Venolia, G. (2009). The secret life of bugs: going past the errors and omissions in software repositories. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering* (pp. 298–308). IEEE Computer Society.

Art and Picture Collection, The New York Public Library. Butterflies (detail). Retrieved from http://digitalcollections.nypl.org/items/510d47e1-26eb-a3d9-e040-e00a18064a99.

Art and Picture Collection-b, The New York Public Library. (1897). Schutzeinrichtungen Ii. Retrieved from http://digitalcollections.nypl.org/items/510d47e1-2a6f-a3d9-e040-e00a18064a99

Avižienis, A., Laprie, J. C., & Randell, B. (2004). Dependability and its threats: a taxonomy. In R. Jacquart (Ed.), *Building the Information Society* (Vol. 156, pp. 91–120). Springer Boston.

Baker, S. J. (n.d.). The Contribution of the Cardboard Cutout Dog to Software Reliability and Maintainability. Retrieved 23 July 2015, from http://www.sjbaker.org/humor/cardboard_dog.html

Ball, L. J., & Ormerod, T. C. (2000). Putting ethnography to work: the case for a cognitive ethnography of design. *Int. J. Hum.-Comput. Stud.*, *53*(1), 147–168.

Bannon, L., Schmidt, K., & Wagner, I. (2011). Lest we forget. In *ECSCW 2011: Proceedings of the 12th European Conference on Computer Supported Cooperative Work, 24-28 September 2011, Aarhus Denmark* (pp. 213–232). Springer.

Barker, C. (2007, November 22). The top 10 IT disasters. *ZDnet*. Retrieved from http://www.zdnet.co.uk/news/it-at-work/2007/11/22/the-top-10-it-disasters-of-all-time-39290976/

Basili, V. R., & Perricone, B. T. (1984). Software errors and complexity: an empirical investigation. *Communications of the ACM*, *27*(1), 42–52. http://doi.org/http://doi.acm.org/10.1145/69605.2085

Behar, R. (1997). *The vulnerable observer: Anthropology that breaks your heart*. Beacon Press.

Bertolino, A., & Strigini, L. (1998). Assessing the risk due to software faults: Estimates of failure rate versus evidence of perfection. *Software Testing, Verification and Reliability*, *8*(3), 155–166.

Beynon-Davies, P. (1997). Ethnography and information systems development: Ethnography of, for and within is development. *Information and Software Technology*, *39*(8), 531 – 540.

Bogdanich, W. (2010). Radiation Offers New Cures, and Ways to Do Harm. *The New York Times*, *n.a.* Retrieved from http://www.nytimes.com/2010/01/24/health/24radiation.html?pagewanted=8

Bowdidge, R. W., & Griswold, W. G. (1997). How software engineering tools organize programmer behavior during the task of data encapsulation. *Empirical Software Engineering*, *2*(3), 221–267.

Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the 27th international conference on Human factors in computing systems* (pp. 1589–1598). ACM.

Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology*, *3*(2), 77–101.

Briand, L. C., Labiche, Y., & Sun, H. (2003). Investigating the use of analysis contracts to improve the testability of object-oriented code. *Software: Practice and Experience*, *33*(7), 637–672.

Brodbeck, F. C., Zapf, D., Prümper, J., & Frese, M. (1993). Error handling in office work with computers: A field study. *Journal of Occupational and Organizational Psychology*, *66*(4), 303–317.

Brewer, W. (n.d.). Schemata. In *Encylopedia of Cognitive Science*. MIT. Retrieved from http://ai.ato.ms/MITECS/Entry/brewer1.html

Brooks, F. P. (1995). *The mythical man-month (anniversary ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Butler, S., Wermelinger, M., Yu, Y., & Sharp, H. (2010). Exploring the influence of identifier names on code quality: an empirical study. Retrieved from http://oro.open.ac.uk/19224/

Buxton, J. N., & Randell, B. (1970). *Software Engineering Techniques: Report on a conference sponsored by the NATO Science Committee Rome, Italy, 27th to 31st October 1969*. Scientific Affairs Division NATO Brussels 39 Belgium: NATO Science Committee. Retrieved from http://homepages.cs.ncl.ac.uk/brian.randell/NATO/

Capretz, L. F. (2014). Bringing the Human Factor to Software Engineering. *IEEE Software*, *31*(2), 104–104. https://doi.org/10.1109/MS.2014.30

Cataldo, M., Mockus, A., Roberts, J. A., & Herbsleb, J. D. (2009). Software dependencies, work dependencies, and their impact on failures. *Software Engineering, IEEE Transactions on*, *35*(6), 864 –878.

Charette, R. N. (2005). Why software fails. *IEEE Spectrum*, *42*(9), 42–49.

Crabtree, A., Tolmie, P., & Rouncefield, M. (2012). *Doing design ethnography*. Springer.

Crandall, B., Klein, G. A., & Hoffman, R. R. (2006). *Working minds: A practitioner's guide to cognitive task analysis*. The MIT Press.

Cross, N. (2001). Design cognition: Results from protocol and other empirical studies of design activity. *Design Knowing and Learning: Cognition in Design Education*, 79–103.

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM*, *31*(11), 1268–1287.

Dalcher, D., & Tully, C. (2002). Learning from failures. *Software Process: Improvement and Practice*, *7*(2), 71–89.

de Souza, C., Redmiles, D., Cheng, L., Millen, D., & Patterson, J. (2004). Sometimes you need to see through walls: A field study of application programming interfaces. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work* (pp. 63–71). Chicago, Illinois, USA: ACM.

Dijkstra, E. W. (1972). 'The Humble Programmer—1972 Turing Award Lecture. *Communications of the ACM*, *15*(10), 859–866.

Dittrich, Y., Randall, D. W., & Singer, J. (2009). Software engineering as cooperative work. *Computer Supported Cooperative Work*, *18*(5-6), 393–399.

Dix, A. (2003). *CSC221 - Introduction to Software Engineering*. Retrieved from http:// www.comp.lancs.ac.uk/ dixa/teaching/CSC221/

Dix, A., Sas, C., Gomes da Silva, P., McKnight, L., Ormerod, T., & Twidale, M. (2006). Why bad ideas are a good idea. *HCIEd'06*.

Easterbrook, S. (2005). Bugs in the space program: the role of software in systems failure. In *INCOSE International Symposium on Systems Engineering*. Retrieved from http://www.cs.toronto.edu/ sme/presentations/BugsInTheSpaceProgram.pdf

Easterbrook, S., Singer, J., Storey, M. A., & Damian, D. (2008). Selecting empirical methods for software engineering research. *Guide to Advanced Empirical Software Engineering*, 285–311.

Eisenstadt, M. (1993). Tales of debugging from the front lines. In *Empirical Studies of Programmers: Fifth Workshop* (pp. 86–112). Palo Alto, CA: Ablex Publishing Corporation.

Eisenstadt, M. (1997). My hairiest bug war stories. *Communications of the ACM*, *40*(4), 30–37.

Ekstedt, M., Johnson, P., & Jacobson, I. (2012). Where's the Theory for Software Engineering? *IEEE Software*, *29*(5), 96.

Endres, A. (1975). An analysis of errors and their causes in system programs. In *Proceedings of the International Conference on Reliable Software* (pp. 327–336). ACM.

Eraut, M. (1994). *Developing professional knowledge and competence*. Psychology Press.

Falzon, M. A. (2009). *Multi-sited ethnography: theory, praxis and locality in contemporary research*. Ashgate Publishing, Ltd.

Ferguson, E. S. (1992). *Engineering and the mind's eye*. MIT Press.

Flanagan, J. C. (1954). The critical incident technique. *Psychological Bulletin*, *51*(4), 327.

Flor, N. V. (1998). Side-by-side collaboration: A case study. *International Journal of Human-Computer Studies*, *49*(3), 201–222.

Flor, N. V., & Hutchins, E. L. (1991). A case study of team programming during perfective software maintenance. In *Empirical studies of programmers: Fourth workshop* (p. 36). Intellect Books.

Forsythe, D. E. (1999). 'It's just a matter of common sense': Ethnography as invisible work. *Computer Supported Cooperative Work (CSCW)*, *8*(1-2), 127–145.

Freeman, J. T., Riedl, T. R., Weitzenfeld, J. S., Klein, G. A., & Musa, J. D. (1991). Instruction for software engineering expertise. In *Proceedings of the SEI Conference on Software Engineering Education* (pp. 271–282). London, UK, UK: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=648325.754778

Frese, M., & Zapf, D. (1994). Action as the core of work psychology: A German approach. *Handbook of Industrial and Organizational Psychology*, *4*, 271–340.

Garfinkel, S. (2005). History's Worst Software Bugs. Retrieved from http://www.wired.com/software/coolapps/news/2005/11/69355

Geertz, C. (2000). 'From the native's point of view': on the nature of anthropological understanding. In *Local Knowledge: Further Essays in Interpretive Anthropology* (pp. 55–72).

Graziotin, D., Wang, X., & Abrahamsson, P. (2015). How do you feel, developer? An explanatory theory of the impact of affects on programming performance. *PeerJ Computer Science*, *1*, e18. https://doi.org/10.7717/peerj-cs.18.

Guindon, R. (1990). Knowledge exploited by experts during software system design. *International Journal of Man-Machine Studies*, *33*(3), 279–304.

Guindon, R., Krasner, H., & Curtis, B. (1987). Breakdowns and processes during the early activities of software design by professionals. In *Empirical Studies of Programmers: Second Workshop. Norwood, NJ: Ablex Publishing Corporation, Lawrence Erlbaum Associates* (pp. 65–82).

Guo, P. J., Zimmermann, T., Nagappan, N., & Murphy, B. (2011). Not my bug! and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work* (pp. 395–404). ACM.

Gusfield, J. (1976). The literary rhetoric of science: comedy and pathos in drinking driver research. *American Sociological Review*, *41*(1), 16–34.

Hammersley, M. (2003). Recent radical criticism of interview studies: any implications for the sociology of education? *British Journal of Sociology of Education*, *24*(1), pp. 119–126.

Hammersley, M., & Atkinson, P. (2007). *Ethnography: Principles in practice*. Routledge.

Hanebutte, N., & Oman, P. W. (2005). Software vulnerability mitigation as a proper subset of software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, *17*(6), 379–400.

Heath, C., Hindmarsh, J., & Luff, P. (2010). *Video in qualitative research: Analysing social interaction in everyday life*. Sage Publications Ltd.

Higgins, A. (2007). Code talk'in soft work. *Ethnography*, *8*(4), 467–484.

Hoare, C. A. R. (1996). How did software get so reliable without proof? In *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods* (p. 17). Springer-Verlag.

Hofmann, D., & Frese, M. (2011). *Errors in organizations*. Routledge.

Hollnagel, E. (1983). Human error. In *Position paper for NATO conference on human error. Bellagio, Italy*.

Hollnagel, E. (1998). *Cognitive reliability and error analysis method (CREAM)*. Elsevier.

Hollnagel, E. (2011). When things go wrong: failures as the flip side of successes. In D. A. Hofmann & M. Frese (Eds.), *Errors in organizations* (pp. 225–244). Routledge.

Hollnagel, E., & Amalberti, R. (2001). The emperor's new clothes: Or whatever happened to 'human error'. In *Proceedings of the 4th international workshop on human error, safety and systems development* (pp. 1–18).

Hollnagel, E., Woods, D. D., & Leveson, N. (Eds.). (2006). *Resilience engineering: concepts and precepts*. Aldershot, England; Burlington, VT: Ashgate.

Holmes, J. (2007). Making humour work: Creativity on the job. *Applied Linguistics*, *28*(4), 518.

Horst, C. (2009). Expanding sites: the question of 'depth' explored. In M. A. Falzon (Ed.), *Multi-sited ethnography: theory, praxis and locality in contemporary research* (pp. 119–134). Ashgate Publishing, Ltd.

Huang, F., Liu, B., & Huang, B. (2012). A taxonomy system to identify human error causes for software defects. In *The 18th international conference on reliability and quality in design*. Retrieved from https://www.researchgate.net/profile/Fuqun_Huang/publication/270160662_A_Taxonomy_System_to_Identify_Human_Error_Causes_for_Software_Defects/links/54a198dd0cf256bf8baf75d1.pdf

Huang, F., Liu, B., Song, Y., & Keyal, S. (2014). The links between human error diversity and software diversity: Implications for fault diversity seeking. *Science of Computer Programming*, *89*, 350–373.

Hughes, J., & Parkes, S. (2003). Trends in the use of verbal protocol analysis in software engineering research. *Behaviour & Information Technology*, *22*(2), 127–140.

Ince, D. (2010). *Victoria climbie, baby p and the technological shackling of british childrens' social work* (No. 2010/01). Open University.

Jordan, B., & Henderson, A. (1995). Interaction analysis: Foundations and practice. *Journal of the Learning Sciences*, *4*(1), 39–103.

Keith, N., & Frese, M. (2005). Self-regulation in error management training: emotion control and metacognition as mediators of performance effects. *Journal of Applied Psychology*, *90*(4), 677.

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K., & Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, *28*(8), 721–734.

Kling, R. (1994). Reading 'All about' computerization: How genre conventions shape nonfiction social analysis. *The Information Society*, *10*, 147–172.

Kling, R., & Courtright, C. (2003). Group behavior and learning in electronic forums: A sociotechnical approach. *The Information Society*, *19*(3), 221–235.

Kling, R., McKim, G., & King, A. (2003). A bit more to it: scholarly communication forums as socio-technical interaction networks. *Journal of the American Society for Information Science and Technology*, *54*(1), 47–67.

Knoblauch, H. (2005). Focused ethnography. In *Forum Qualitative Sozialforschung/ Forum: Qualitative Social Research* (Vol. 6).

Knoblauch, H., & Schnettler, B. (2012). Videography: analysing video data as a 'focused'ethnographic and hermeneutical exercise. *Qualitative Research*, *12*(3), 334–356.

Knoblauch, H., & Tuma, R. (2011). Videography. An interpretative approach to video-recorded micro-social interaction. *The SAGE Handbook of Visual Research Methods*, 414–430.

Knuth, D. E. (1989). The errors of TEX. *Software: Practice and Experience*, *19*(7), 607–685.

Ko, A. J., DeLine, R., & Venolia, G. (2007). Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering* (pp. 344–353). IEEE Computer Society.

Ko, A. J., & Myers, B. (2008). Debugging reinvented. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on* (pp. 301–310). IEEE.

Ko, A. J., & Myers, B. A. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, *16*(1), 41–84.

Koenemann-Belliveau, J., Carroll, J. M., Rosson, M. B., & Singley, M. K. (1994). Comparative usability evaluation: Critical incidents and critical threads. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Celebrating Interdependence*, 245–251.

Krasner, G. (1983). *Smalltalk-80: bits of history, words of advice*. Addison-Wesley Longman Publishing Co., Inc.

Kristoffersen, S. (2006). Designing a program. programming the design. *TeamEthno-Online Issue*, *2*, 34–51.

Lammers, S. (1986). *Programmers at work*. Harper & Row Publishers, Inc. New York, NY, USA.

LaToza, T. D., & Myers, B. A. (2010). On the importance of understanding the strategies that developers use. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering* (pp. 72–75). ACM.

LaToza, T. D., & Myers, B. A. (2011). Designing useful tools for developers. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools* (pp. 45–50). ACM.

Laurier, E., & Philo, C. (2006). Natural Problems of Naturalistic Video Data. In H. Knoblauch, B. Schnettler, J. Raab, & H.-G. Soeffner (Eds.), *Video Analysis. Methodology and Methods.* Frankfurt am Main: Peter Lang.

Lave, J. (1988). *Cognition in practice: Mind, mathematics and culture in everyday life*. Cambridge University Press.

Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., & Fleming, S. D. (2013). How programmers debug, revisited: An information foraging theory perspective. *Software Engineering, IEEE Transactions on*, *39*(2), 197–215.

Le Coze, J. C. (2015). Reflecting on Jens Rasmussen's legacy. A strong program for a hard problem. *Safety Science*, *71, Part B*(0), 123 – 141.

Leszak, M., Perry, D. E., & Stoll, D. (2002). Classification and evaluation of defects in a project retrospective. *The Journal of Systems & Software*, *61*(3), 173–187.

Leveson, N. G., & Turner, C. S. (1993). Investigation of the Therac-25 accidents. *IEEE Computer*, *26*(7), 18–41.

Levy, M., Salvadori, M., & Woest, K. (2002). *Why buildings fall down: how structures fail*. WW Norton & Company.

Lewis, C., & Norman, D. A. (1986). Designing for Error. In *User Centered System Design*. Erlbaum Associates, Inc.

Lopez, T., Petre, M., & Nuseibeh, B. (2012a). Getting at ephemeral flaws. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2012 5th International Workshop* (pp. 90–92). IEEE.

Lopez, T., Petre, M., & Nuseibeh, B. (2012b). Thrashing, tolerating and compromising in software development. In *Psychology of Programming Interest Group Annual Conference (PPIG-2012), London Metropolitan University, UK*. London Metropolitan University, UK: London Metropolitan University, UK, London Metropolitan University.

Lopez, T., Petre, M., & Nuseibeh, B. (2015). Active Error: Examining Error Detection and Recovery in Software Development. Presented at the Psychology of Programming

Interest Group, Work-in-Progress Meeting, 2015., School of Science and Technology, Middlesex University. London, UK.

Lutters, W. G., & Seaman, C. B. (2007). Revealing actual documentation usage in software maintenance through war stories. *Information and Software Technology*, *49*(6), 576 – 587.

Magalhães, J., von Staa, A., & de Lucena, C. J. . (2009). Evaluating the recovery-oriented approach through the systematic development of real complex applications. *Software: Practice and Experience*, *39*(3), 315–330.

Mahoney, M. S. (2008). What makes the history of software hard. *IEEE Annals of the History of Computing*, *1*(3), 8–18.

Markus, M. L. (1994). Finding a happy medium: Explaining the negative effects of electronic communication on social life at work. *ACM Transactions on Information Systems (TOIS)*, *12*(2), 119–149.

Martin, M. D., & Sommerville, I. (2004). Patterns of cooperative interaction: Linking ethnomethodology and design. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *11*(1), 59–89.

McDonnell, J. (2012). Accommodating disagreement: A study of effective design collaboration. *Design Studies*, *33*(1), 44–63.

McGinn, M. K. (2008). Secondary data. In L. M. Given (Ed.), *The Sage encyclopedia of qualitative research methods*. Sage Publications.

Miyake, N. (1986). Constructive interaction and the iterative process of understanding. *Cognitive Science*, *10*(2), 151–177.

Narayan, K. (2012). *Alive in the writing: Crafting ethnography in the company of Chekhov*. University of Chicago Press.

Naur, P., & Randell, B. (1969). *Software Engineering: Report on a conference sponsored by the NATO Science Committee Garmisch, Germany, 7th to 11th October 1968*. Scientific Affairs Division NATO Brussels 39 Belgium: NATO Science Committee. Retrieved from http://homepages.cs.ncl.ac.uk/brian.randell/NATO/

Nickerson, J. V., & Yu, L. (2010). *'There's Actually a Car' Perspective taking and evaluation in software-intensive systems design conversations*.

Norman, D. A. (1981). Categorization of action slips. *Psychological Review*, *88*(1), 1–15.

Norman, D. A. (1983). Design rules based on analyses of human error. *Communications of the ACM*, *26*(4), 258.

Norman, D. A. (2002). *The design of everyday things*. Basic Books.

Norman, D. A., & Shallice, T. (1986). Attention to action. In R. J. Davidson, G. E. Schwartz, & D. Shapiro (Eds.), *Consciousness and Self-Regulation* (pp. 1–18). Springer US.

North, D. (n.d.). Introducing BDD. Retrieved from http://dannorth.net/introducing-bdd/

Nuseibeh, B. (1997). Ariane 5: Who dunnit? *IEEE Software*, *14*, 15–16. https://doi.org/10.1109/MS.1997.589224

Orlikowski, W. J. (1992). Learning from notes: Organizational issues in groupware implementation. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work* (pp. 362–369). ACM.

Orlikowski, W. J., & Gash, D. C. (1994). Technological frames: making sense of information technology in organizations. *ACM Transactions on Information Systems (TOIS)*, *12*(2), 174–207.

Orlikowski, W. J., & Iacono, C. S. (2001). Research commentary: Desperately seeking the 'IT' in IT research—A call to theorizing the IT artifact. *Information Systems Research*, *12*(2), 121–134.

Orr, J. E. (1986). Narratives at work: Story telling as cooperative diagnostic activity. In *Proceedings of the 1986 ACM conference on Computer-supported cooperative work* (pp. 62–72). ACM.

Pascal, C. (n.d.). The teddy bear principle in programming. Retrieved from http://compas-pascal.blogspot.de/2007/12/teddy-bear-principle

Päivärinta, T., & Smolander, K. (2015). Theorizing about software development practices. *Science of Computer Programming*, *101*, 124–135. https://doi.org/10.1016/j.scico.2014.11.012

Pennington, N., & Grabowski, B. (1990). The tasks of programming. *Hoc et Al*, *307*, 45–62.

Perry, D. E. (2010). Where do most software flaws come from? In A. Oram & G. Wilson (Eds.), *Making Software: What Really Works, and Why We Believe It* (pp. 453–494). O'Reilly Media, Inc.

Perry, D. E., & Evangelist, W. M. (1985). An empirical study of software interface faults. *Proceedings of the International Symposium on New Directions in Computing*, 32–38.

Perry, D. E., & Evangelist, W. M. (1987). An empirical study of software interface faults — an update. In *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences* (Vol. II, pp. 113–126).

Perry, D. E., & Stieg, C. (1993). Software faults in evolving a large, real-time system: a case study. In *Proceedings of the 4th European Software Engineering Conference on Software Engineering* (pp. 48–67). Springer-Verlag.

Perrow, C. (1984). *Normal accidents: living with high-risk technologies*. New York: Basic Books.

Plonka, L., Sharp, H., & Van der Linden, J. (2012). Disengagement in pair programming: does it matter? In *Software Engineering (ICSE), 2012 34th International Conference on* (pp. 496–506). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6227166

Prior, J. (2011). *Everyday practices of agile software developers*. University of Technology Sydney.

Pugh, W. (2009). Mistakes that matter. In *JavaOne Conference*. University of Maryland. Retrieved from http://www.cs.umd.edu/ pugh/MistakesThatMatter.pdf

Pullum, L. L. (2001). *Software fault tolerance techniques and implementation*. Norwood, MA, USA: Artech House, Inc.

Randell, B. (1998). Dependability-a unifying concept. In *Proceedings of the Conference on Computer Security, Dependability, and Assurance: From Needs to Solutions*. IEEE Computer Society Washington, DC, USA.

Randell, B. (2003). On failures and faults. In K. Araki, S. Gnesi, & D. Mandrioli (Eds.), *FME 2003: Formal Methods* (Vol. 2805, pp. 18–39). Springer Berlin / Heidelberg.

Randell, B. (2007). A computer scientist's reactions to NPfIT. *Journal of Information Technology*, *22*(3), 222–234.

Rasmussen, J. (1985). *Human error data. Facts or fiction?* Roskilde, Denmark: Riso National Laboratory.

Rasmussen, J. (1990). The role of error in organizing behaviour. *Ergonomics*, *33*(10-11), 1185–1199.

Rasmussen, J. (1997). Risk management in a dynamic society: a modelling problem. *Safety Science*, *27*(2), 183–213.

Rasmussen, J., & Jensen, A. (1974). Mental procedures in real-life tasks: a case study of electronic trouble shooting. *Ergonomics*, *17*(3), 293–307.

Rasmussen, J., Nixon, P., & Warner, F. (1990). Human error and the problem of causality in analysis of accidents [and discussion]. *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, *327*(1241), 449–462.

Rasmussen, J., Pejtersen, A. M., & Schmidt, K. (1990). *Taxonomy for cognitive work analysis*. Riso National Laboratory.

Reason, J. (1984). Lapses of attention in everyday life. *Varieties of Attention*, 515–549.

Reason, J. (1990). *Human Error*. New York: Cambridge University Press.

Reason, J. (2004). Beyond the organisational accident: the need for "error wisdom" on the frontline. *Quality and Safety in Health Care*, 13(suppl 2), ii28–ii33.

Reason, J., Hollnagel, E., & Paries, J. (2006). Revisiting the «Swiss cheese» model of accidents. EEC Note No. 13/06. Retrieved from: http://publish.eurocontrol.int/eec/gallery/content/public/document/eec/report/2006/017_Swiss_Cheese_Model.pdf

Reason, J., Manstead, A., Stradling, S., Baxter, J., & Campbell, K. (1990). Errors and violations on the roads: a real distinction? *Ergonomics*, *33*(10-11), 1315–1332.

Riedl, T. R., Weitzenfeld, J. S., Freeman, J. T., Klein, G. A., & Musa, J. D. (1991). What we have learned about software engineering expertise. In *Proceedings of the SEI Conference on Software Engineering Education* (pp. 261–270). London, UK, UK: Springer-Verlag.

Rizzo, A., Bagnara, S., & Visciola, M. (1987). Human error detection processes. *International Journal of Man-Machine Studies*, *27*(5), 555–570.

Rizzo, A., Ferrante, D., & Bagnara, S. (1995). Handling human error. In *Expertise and technology: Cognition & human-computer cooperation* (pp. 195–212).

Rizzo, A., Parlangeli, O., Marchigiani, E., & Bagnara, S. (1996). The management of human errors in user-centered design. *ACM SIGCHI Bulletin*, *28*(3), 114–118.

Robillard, M. P. (2009). What makes APIs hard to learn? Answers from developers. *IEEE Software*, *26(6)*, 24–34.

Robinson, H., Segal, J., & Sharp, H. (2007). Ethnographically-informed empirical studies of software practice. *Information and Software Technology*, *49*(6), 540 – 551.

Rubber duck debugging. (n.d.). In *Wikipedia*. Addison Wesley. Retrieved from https://en.wikipedia.org/wiki/Rubber_duck_debugging

Sachs, P. (1995). Transforming work: collaboration, learning, and design. *Commun. ACM*, *38*(9), 36–44.

Schneidewind, N. F., & Hoffmann, H. M. (1979). An experiment in software error data collection and analysis. *Software Engineering, IEEE Transactions on*, *SE-5*(3), 276 – 286.

Scott, J. (1990). *A matter of record: documentary sources in social research* (Vol. 12). Polity Press Cambridge.

Seaman, C. B., & Basili, V. R. (1997). An empirical study of communication in code inspections. In *Proceedings of the 19th international conference on Software engineering* (p. 106). ACM.

Sellen, A. J. (1994). Detection of everyday errors. *Applied Psychology*, *43*(4), 475–498.

Sharp, H., Robinson, H., & Woodman, M. (2000). Software engineering: community and culture. *Software, IEEE*, *17*(1), 40 –47.

Shaw, M. (2002). 'Self-healing': softening precision to avoid brittleness: position paper for WOSS '02: workshop on self-healing systems. *WOSS '02: Proceedings of the First Workshop on Self-Healing Systems*, 111–114.

Sillito, J., Murphy, G. C., & De Volder, K. (2008). Asking and answering questions during a programming change task. *Software Engineering, IEEE Transactions on*, *34*(4), 434–451.

Sillitti, A., Succi, G., & Vlasenko, J. (2012). Understanding the impact of pair programming on developers attention: a case study on a large industrial experimentation. In *Proceedings of the 2012 International Conference on Software Engineering* (pp. 1094–1101). IEEE Press.

Smith, M. K. (2003). Michael Polanyi and tacit knowledge. Retrieved from http://infed.org/mobi/michael-polanyi-and-tacit-knowledge/

Sözer, H., Tekinerdoğan, B., & Akşit, M. (2009). FLORA: A framework for decomposing software architecture to introduce local recovery. *Software: Practice and Experience*, *39*(10), 869–889. http://doi.org/10.1002/spe.916

Spradley, J. P. (1979). *The Ethnographic Interview*. Wadsworth Publishing Company.

Spradley, J. P. (1980). *Participant Observation*. Harcourt Brace College Publishers.

Storey, M. A., Ryall, J., Bull, R. I., Myers, D., & Singer, J. (2008). TODO or to bug. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on* (pp. 251–260). IEEE.

Suchman, L. A. (1987). *Plans and situated actions: the problem of human-machine communication*. Cambridge university press.

Svenonius, E. (2000). *The intellectual foundation of information organization*. MIT press.

Taylor, R. N., & van der Hoek, A. (2007). Software design and architecture the once and future focus of software engineering. In *International Conference on Software Engineering* (pp. 226–243). IEEE Computer Society Washington, DC, USA.

Than, T. T., Jackson, M., Laney, R., Nuseibeh, B., & Yu, Y. (2009). Are your lights off? Using problem frames to diagnose system failures. *Requirements Engineering, IEEE International Conference on*, *0*, v–ix.

Turkle, S. (2005). *The second self: Computers and the human spirit*.

Van Maanen, J. (2011). *Tales of the Field: On Writing Ethnography* (Second Edition). University of Chicago Press.

Vinson, N. G., & Singer, J. (2008). A practical guide to ethical research involving humans. In *Guide to Advanced Empirical Software Engineering* (pp. 229–256). Springer.

Walia, G. S., Carver, J. C., & Bradshaw, G. (2015). Workshop on applications of human error research to improve software engineering (WAHESE 2015). In *Proceedings of the 37th International Conference on Software Engineering-Volume 2* (pp. 1019–1020). IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2819259

Weatherbee, T. G. (2009). Critical incident case study. In A. J. Mills, G. Durepos, & E. Wiebe (Eds.), *Encyclopedia of case study research* (Vol. 1, pp. 257–248). Sage Publications.

Weick, K. E. (1990). The vulnerable system: An analysis of the Tenerife air disaster. *Journal of Management*, *16*(3), 571–593.

Weinberg, G. M. (1998). *The psychology of computer programming (silver anniversary ed.)*. New York, NY, USA: Dorset House Publishing Co., Inc.

Weitzenfeld, J. S., Riedl, T. R., Freeman, J. T., Klein, G. A., & Musa, J. D. (1991). Knowledge elicitation for software engineering expertise. In *Proceedings of the SEI Conference on Software Engineering Education* (pp. 283–296). London, UK, UK: Springer-Verlag.

Wildman, D. (1995). Getting the most from paired-user testing. *Interactions*, *2*(3), 21–27. http://doi.org/10.1145/208666.208675

Winograd, T., & Flores, F. (1987). *Understanding Computers and Cognition*. Addison-Wesley.

Woods, D. D., & Cook, R. I. (1999). Perspectives on human error: Hindsight biases and local rationality. *1999), Handbook of Applied Cognition*.

Woods, D. D., & Cook, R. I. (2003). Mistaking error. *Patient Safety Handbook*, 95–108.

Woods, D. D., Johannesen, L. J., Cook, R. I., & Sarter, N. B. (1994). *Behind human error: Cognitive systems, computers and hindsight*. DTIC Document.

Xu, S., & Rajlich, V. (2005). Dialog-based protocol: an empirical research method for cognitive activities in software engineering. In *Empirical Software Engineering, 2005. 2005 International Symposium on* (p. 10–pp). IEEE.

Zapf, D., Brodbeck, F. C., Frese, M., Peters, H., & Prümper, J. (1992). Errors in working with office computers: A first validation of a taxonomy for observed errors in a field setting. *International Journal of Human-Computer Interaction*, *4*(4), 311–339.

Zapf, D., Maier, G. W., Rappensperger, G., & Irmer, C. (1994). Error detection, task characteristics, and some consequences for software design. *Applied Psychology*, *43*(4), 499–520.

Zorich, D. (2008). *A survey of digital humanities centers in the united states*. Council on Library and Information Resources.

Zou, F. (2003). A change-point perspective on the software failure process. *Software Testing, Verification and Reliability*, *13*(2), 85–93.

Appendices

A. Conventions and Tools

A.1 Transcription

Audio recorded interviews, design video and programming videos were transcribed using the same basic transcription conventions, defined to capture details of speech and interaction. Conventions were adjusted to meet requirements of different media. For example, audio interviews taken for Chapter 7 do not permit the transcription of behaviour or action that took place while work was in progress, while the design videos (Chapter 5. At the Drawing Board) do. The programming videos (Chapter 6) included a fair amount of reading of content on the screen which was indicated where possible using quotes in the transcription and a note indicating that the developer was Reading or Narrating.

Conventions were adapted from two sources. In "Making Humour Work: Creativity on the Job", Holmes analysed everyday workplace interaction to examine claims that humour is associated with creativity in the work place (Holmes, 2007). Transcription conventions given in the IASRDR 2009 tutorial on analysing design meetings also made use of Holmes' conventions and were used to develop notation.

Transcriptions were made using the software Transcriptions (https: //code.google. com/ p/transcriptions/) (R.I.P.) and its commercial counterpart f5, available at: http: //www. audiotranskription. de. Timestamps were entered into the texts following the conventions of these programs. Timestamps were noted at four or five minute increments, but have also been entered during analysis to indicate the start and end of incidents and to mark other points of interest.

Transcription Conventions	
<u>wow</u>	Underlining indicate emphatic stress
" "	Quotes indicate when the words a speaker makes mirror text being written on a screen or read from a specification.
+	+ Symbols indicate pauses. Pauses were not timed, however longer pauses are represented with multiple + symbols (generated using a "one one thousand, two one thousand count").

.../... /...	Interjection (e. g. of assent) or simultaneous speech, placed at approximate point of occurrence.
[Reads from prompt]	Transcriber's comments regarding action on the screen. Comments may include reference to the name of a corresponding image file that depicts a scene from the video at that point.
[Gest: moves hand]	Comment describing a gesture that is made on-screen.
[WB: draws a box]	Comment about activity at the whiteboard.
[Voce: Character| Falsetto]	Comment indicating that the speaker is using a voice. Start and end points are indicated with:.
(Laughs)	Paralinguistic activity is described within parentheses, may also occur within comments
(inaudible)	Inaudible text
(so then we)	Best guess about inaudible text
--	Cut-off utterance. Also used to indicate where speaker 'jumps in'
	Trailing off
___	Underscores are used to anonymize data.

Table A.1: Transcription conventions.

A.2 Signalling Devices

The critical decision method protocol for collecting retrospective accounts of work suggests that ***verbal signals*** may be given during interviews that will indicate decision points or developments in a problem solving sequence (Crandall et al., 2006). This was also found during analysis to be true of accounts given by pairs.

In the data analysed for the studies reported in Chapters 5, 6 and 7, verbal signals indicated how an insight was perceived at the time of the incident or interview, or indicated a way in which the experience unfolded that ran counter to the informant's former or desired experience. Verbal signals were also found to highlight the detection of a problem, or indicate a resolution. They indicated options that an informant considered, or revealed constraints on action brought about by policies or practices in the broader work environ-

ment. Broad characteristics of signals are given below, while a sample of verbal signals collected from the data can be read in Table A.2.

Repetition often indicates the presence of an incident to the analyst, particularly in regards to conceptual or design issues that do not have clearly demarcated, corresponding material actions. Sometimes repetition of a single phrase or detail may “name” the problem for the analyst. Verbal signals related to repetition include language that indicated disagreement (e. g. "I don't think so") or lack of understanding (e. g "I don't know"), and may have been accompanied by expressions of lack of confidence in the ideas being expressed.

Signals may also be **gestural.** Lack of confidence might be signalled by repeated turns away from a whiteboard as if to seek assent from the partner and the corresponding provision of assent in the form of paralinguistic utterances (e. g. "mm hmm", "yeah").

Signals may be **reactive**. Informants on the videos do not always indicate what they are thinking or why they perform some actions. The interview data I collected was likewise at times a bit too "artful" (Hammersley, 2003). It was sometimes evident that informants said what they thought was wanted for the research, or relayed a desired behaviour that did not match other evidence. The developers were at also times reactive to the experience of being filmed (Laurier & Philo, 2006). In the quote that follows, Marcus displays a clear awareness of the recording.

> "As soon as I know I'm recorded, I start talking a lot. I should have been a DJ." - (Marcus, Ep. 1, 10:30).

It is not possible to say that Marcus’ behaviour was different than it would have been if he hadn’t been recorded, but it, along with other indicators served as signalling devices during analysis, reminders that the videos, while naturalistic, were not entirely natural

Verbal Signals (Selected)	
Reasoning	it took me a moment to realise poking around in the dark”
Naming Problems:	Sticking point That’s a funniness
Worry/ Concern:	I’m quite wary of screwing things up It doesn’t feel right”

Serendipity	Suddenly it worked That rang a bell straight away I have been lucky Touch wood
Preferred ways of working:	I don't like to The way I usually do that is Around here
Interestingness	That's interesting Strange, very strange That's funny
Prior experience	The problem was much simpler before The last time I've seen it before

Table A.2. Verbal signals used to develop sequences of error handling. This list is representative, not comprehensive.

B. Notes on At the Drawing Board

The study *At the Drawing Board* is reported in Chapter 5. It drew upon data collected at Site A. For an overview of sites, see Chapter 4, Section 4.3. Other detail about data collection and analysis are reported in Chapter 4, Section 4.4.1, and in Chapter 5, Section 5.2.

B.1 Columnar Analysis

Columnar transcription conventions were adapted from *Interaction analysis: Foundations and practice* (Jordan & Henderson, 1995). Data included in the columnar transcription was taken from the full transcription of dialogue, but segmented and organised to facilitate analysis. Multiple time-stamped entries from the rich transcription were at times grouped into a single exchange, represented within a table row. Guindon's 'Kinds of Knowledge' (1990) (see also Section B.3 for a summary) were used to annotate the 'analysis' column for individual exchanges. The schema for the columnar transcription is

given in Table B.1.1, while an excerpt from a transcription is provided in Table B.2.2, below.

Episode Number (~length of the episode in minutes) Summary: A brief two or three sentence explanation of the episode; comparable to the information supplied in the Content Log described by Jordan and Austin (1995)				
Verbal	Gesture/Action	Whiteboard	Reference to prompt	Analysis
#00:41:23.0# Verbal data of the exchange is included here, underneath a timestamp and with cross references [n] to gestures, actions, whiteboard activity, or references by designers to the design prompt.	[n] note on gesture	[n] note on whiteboard activity	[n] note on use of prompt	Commentary on design activity given here

Table B.1.1: Columnar transcription.

In Table B.1.2 that follows, an excerpt is given of the columnar analysis made for the incident reported in Chapter 5, Section 5.3.3.

Episode 1 (< 1 min.) F wonders how the results of the simulation are quantified. Two issues are intertwined: one, how simulations are configured, reported, and saved by the user in the user interface, and; two, what represents a simulation, what constitutes success or failure. The first issue prompts the designers to consider immediate implications of managing simulations. The second relates to how factors such as speed, distance, and car density on roads should optimally be combined by students to produce simulations.				
Verbal	Gesture/Action	White-board	Prompt	Analysis

#00:08:22.1# F: Well, so one is you want to change the layout of the map, M: um hmm F: two is you want to change the parameters you gave it in terms of speeds and timings, right? And three, you want to run it, meaning little dots are moving, showing you how the traffic is flowing. And what does that mean? [1] F: How do you? What kind of metric do you get back to tell you this is working, you know? How do you assess the success M: Yeah it kind of feels like (inaudible)/of the timing?	[1] M draws a pattern of lines on the Drawing Area			F enumerates functions the program should support. problem framing ("you want..."), with reference to the solution space ("meaning little dots are moving, showing you..."). It marks the introduction of the "what does it's working mean" difficulty. Asking questions: "what does that mean? How do you? What kind of metric do you get back to tell you this is working, you know? How do you assess the success?"
#00:08:58.0# M: It's nice in the simulation to be able to watch what's going on here, and you need kind of a summary area to kind of tell you[1] [2] F: Yes, mm hmm. [3] M: to kind of tell you what your settings-- F: A dashboard. M:-- are for the individual intersections, and what kind of effect, like how much is the traffic backup at this light F: Exactly. M: or what's the average wait−time at this light [4] . So in terms of objects that they need to deal with, there's, are we going to call them streets or roads [5] ?	[1] M waves hand over an area to the right of the grid he has been drawing. [3] F, with hand toward the area to the right he has added [4] M shoulder shrugs	[2] M adds a rectangular block to the right-hand side of the board	[5] Refers to design prompt	M moves into the solution space with a UI feature: the addition of the Summary Area (Sect. 3.3). The external representation is low-fi, just a box, but he verbally notes the kinds of behaviours this section will support. His reference to "what kind of effect" is the only reference to the problem raised by F. Note that M makes clear the transition to solution with a hand gesture over the right hand portion of the screen that she mirrors. F introduces the concept 'dashboard' as a way to describe the purpose of the summary area, but this is not picked up by M. M introduces a design strategy (Sect 3.4) here with discussion about "objects that they need to deal with". These are documented as a list using a blue marker.

Table B.1.2 Excerpt of columnar analysis. This is an excerpt from the incident analysed in Chapter 5, Section 5.3.3

B.2 Design Prompt

Following is the design prompt used by Kasia and Bill in the Amberpoint design session. The prompt was written by the organisers of the "Studying Professional Software Design" (SPSD) workshop; it was issued to designers at the time of participation.

Design Prompt: Traffic Signal Simulator

Problem Description
For the next two hours, you will be tasked with designing a traffic flow simulation pro- gram.

Your client for this project is Professor E, who teaches civil engineering at UCI. One of the courses she teaches has a section on traffic signal timing, and according to her, this is a particularly challenging subject for her students. In short, traffic signal timing involves determining the amount of time that each of an intersection's traffic lights spend being green, yellow, and red, in order to allow cars in to flow through the intersection from each direction in a fluid manner. In the ideal case, the amount of time that people spend waiting is minimized by the chosen settings for a given intersection's traffic lights. This can be a very subtle matter: changing the timing at a single intersection by a couple of seconds can have far-reaching effects on the traffic in the surrounding areas.

There is a great deal of theory on this subject, but Professor E. has found that her students find the topic quite abstract. She wants to provide them with some software that they can use to "play" with different traffic signal timing schemes, in different scenarios. She anticipates that this will allow her students to learn from practice, by seeing first-hand some of the patterns that govern the subject.

Requirements
The following broad requirements should be followed when designing this system:

1. Students must be able to create a visual map of an area, laying out roads in a pattern of their choosing. The resulting map need not be complex, but should allow for roads of varying length to be placed, and different arrangements of intersections to be created. Your approach should readily accommodate at least six intersections, if not more.

2. Students must be able to describe the behavior of the traffic lights at each of the intersections. It is up to you to determine what the exact interaction will be, but a variety of sequences and timing schemes should be allowed. Your approach should also be able to accommodate left-hand turns protected by left-hand green arrow lights. In addition:

1. Combinations of individual signals that would result in crashes should not be allowed.

2. Every intersection on the map must have traffic lights (there are not any stop signs, overpasses, or other variations). All intersections will be 4-way: there are no "T" intersections, nor one-way roads.

3. Students must be able to design each intersection with or without the option to have sensors that detect whether any cars are present in a given lane. The intersection's lights' behavior should be able to change based on the input from these sensors, though the exact behavior of this feature is up to you.

3. Based on the map created, and the intersection timing schemes, the students must be able to simulate traffic flows on the map. The traffic levels should be conveyed visually to the user in a real-time manner, as they emerge in the simulation. The current state of the intersections' traffic lights should also be depicted visually, and updated when they change. It is up to you how to present this information to the students using your program. For example, you may choose to depict individual cars, or to use a more abstract representation.

4. Students should be able to change the traffic density that enters the map on a given road. For example, it should be possible to create a busy road, or a seldom- used one, and any variation in between. How exactly this is declared by the user and depicted by the system is up to you.

Broadly, the tool should be easy to use, and should encourage students to explore multiple alternative approaches. Students should be able to observe any problems with their map's timing scheme, alter it, and see the results of their changes on the traffic patterns.

This program is not meant to be an exact, scientific simulation, but aims to simply illustrate the basic effect that traffic signal timing has on traffic. If you wish, you may assume that you will be able to reuse an existing software package that provides relevant mathematical functionality such as statistical distributions, random number generators, and queuing theory.

You may add additional features and details to the simulation, if you think that they would support these goals.

Your design will primarily be evaluated based on its elegance and clarity – both in its overall solution and envisioned implementation structure.

Desired Outcomes
Your work on this design should focus on two main issues:

1. You must design the interaction that the students will have with the system. You should design the basic appearance of the program, as well as the means by which the user creates a map, sets traffic timing schemes, and views traffic simulations.

2. You must design the basic structure of the code that will be used to implement this system. You should focus on the important design decisions that form the foundation of the implementation, and work those out to the depth you believe is needed.

The result of this session should be: the ability to present your design to a team of soft- ware developers who will be tasked with actually implementing it. The level of competency you can expect is that of students who just completed a basic computer science or software engineering undergraduate degree. You do not need to create a complete, final diagram to be handed off to an implementation team. But you should have an under- standing that is sufficient to explain how to implement the system to competent developers, without requiring them to make many high-level design decisions on their own.

To simulate this hand-off, you will be asked to briefly explain the above two aspects of your design after the design session is over.

Timeline
• 1hourand50minutes:Designsession

• 10 minutes: Break / collect thoughts
• 10 minutes: Explanation of your design

• 10 minutes: Exit questionnaire

B.3 Kinds of Expert Knowledge

In the 1990 paper *Knowledge exploited by experts during software system design*, Raymonde Guindon analysed the specialized knowledge used by software designers when performing early design tasks. Her analysis included information about the kinds of new knowledge generated, the ways in which designers leverage existing knowledge, and a set of heuristics used to seek and select design solutions. Guindon's findings from this paper are extracted and consolidated here into a catalogue that was used as an aid to analysis of early design activities in the SPSD session. They are enumerated according to the section of that paper in which they appear.

Sect. 3.1 Retrieval or simulation of scenarios in the problem domain (the real world). Interwoven with solution development, spoken scenarios are often accompanied by external representations in the form of diagrams with annotations.

Scenarios serve five purposes:

1. Understand given requirements - before problem solving, as a way of confirming understanding of requirements.
2. Understand inferred requirements - upon inferring requirements, as a way of confirming the relevance of the discovery.
3. Solution development - to generate new ideas, to jumpstart progress. When used in this way, the scenarios are used to frame and structure the problem.
4. Discovery (unplanned) of new requirements - used to simulate and evaluate the solution.
5. Discovery (unplanned) of partial solutions - the scenario triggers the recognition of a partial solution.

Sect. 3.2 Requirements elaboration, used to reduce ambiguity inherent in the design prompt and to decrease the range of possible solutions by acting as "simplifying assumptions" (p. 290). Run throughout the design session, structure and frames the problem, and suggests evaluation criteria for solution selection. External representations in the form of lists of notes are used to "keep track" of requirements.

Inferred constraint - unstated in the given requirements, but are inferred as logically necessary based on what is stated, and the designer's own knowledge of the problem domain. They reduce incompleteness and ambiguity in the stated requirements, with direct consequences for the solution. In design sessions, they often result in changes in immediate design goals. That is, the designers shift the focus of their thinking to handle the newly inferred requirement.

Added requirement - a desirable but not necessary requirement for the production of a logically sound design. They reflect preferred evaluation criteria, or rules by which designers signify stopping points.

Sect. 3.3 Design Solutions, the designer's understanding of the solution, and the way this understanding is externally represented The way a solution is decomposed into sub-problems may vary between designers, as may the selection of notational systems for representation. In general, she observed the following uses of external representations:

1. to express the design solution
2. to support mental simulations of the solution in the form of "test cases" based on knowledge of the problem domain.
3. reveal missing information

4. ensure completeness of the solution

Mental simulations uncover various kinds of "bugs" in the solution:

1. inconsistencies within given or inferred requirements
2. inconsistencies between parts of the solution
3. incompleteness of partial solutions in respect to the whole

N. B. Guindon states a fourth, but it seems to be a duplication of an earlier point

Notational systems serve two purposes:

- express the design solution
- tools for developing the solution

Sect. 3.4 Design strategies, methods and notations, that is, the sequence of activities to be performed, as structured by a recognized design method. Examples of design strategies given are: top-down, data structure-oriented and object-oriented structure. Designers can use more than one strategy in a single session, and may also use multiple notational systems.

Sect 3.5 Problem solving and software design schema, or higher order knowledge structures such as divide-and-conquer and generate-and-test. Guindon found that in her data, specialized schema used by designers varied in complexity and granularity. She suggested that the schema is a "complex rule composed of a pattern which specifies the similarities in requirements between different instances of a class of systems (e. g. resource allocation systems)." (p. 296). Schema are selected based on similarities between the current problem and known patterns.

Sect. 3.6 Design heuristics are used by designers in problem structuring and solution generation

1. consider a simpler problem first, then later expand the solution
2. simulate scenarios in the problem domain to acquire more information about the problem structure
3. identify system functions that can be performed nearly independently and divide the system into corresponding subsystems
4. avoid serious mistakes or catastrophes
5. satisfy the most important constraints or requirements first
6. keep the design solution as simple as possible
7. make simplifying assumptions about the requirements
8. keep the solution parts as consistent
9. delay commitment to decision when there is insufficient information; re-examine tentative decisions as new information is acquired.

Sect. 3.7 Preferred evaluation criteria are adopted in order to manage the ill-defined nature of design problems. Designers adopt a "small set of personalized criteria" (p. 298) to guide solution generation and selection. For example, one of her developers adopted high reliability as a criterion. Unstated in the requirements, this criterion was used in schema selection, and thus to reduce the set of possible designs to consider. Other observed criteria included simplicity of solution and simplicity of design process.

C. Notes on At the Desk

The study *At the Desk* is reported in Chapter 6. It drew upon data collected at Site C. For an overview of sites, see Chapter 4, Section 4.3. Other detail about data collection and analysis are reported in Chapter 4, Section 4.4.2, and in Chapter 6, Section 6.2.

C.1. Transcription and Cataloguing

Analysis began with the selection of a data set. A master catalogue was made to track the sixty videos uploaded to a web hosting site. The catalogue documented metadata from the video hosting website, the code repository, and information required for research. It also included information on video quality, notes about the content, and approximate recording date. Twenty episodes were selected for deeper examination, and were transcribed using the conventions noted in appendix A. Notes on Cataloguing follow.

Episodes 1−10: A near-verbatim transcription was created of each episode. A content log was developed to note what happened at regular intervals. The content logs were coded to capture impressions about themes running through the data. The codes were analysed and compared to evidence of themes that emerged in analysis of the design videos reported in Chapter 5 and the first set of interviews taken at Site B. This analysis gave a sense for incident kinds, concentration, and of curiosities in the way the developers talked about them. It also familiarised the researcher with environmental context and working style of the developers.

A catalogue of observable features of incidents was created to include information about the start time, end time and duration of the incident, a brief description with more detailed impressions about the significance of the incidence, the driver, the end result, files involved, a rough identification of the source of the error, and a snippet of dialogue that stood out as capturing the essence of the incident.

Episode 11-20: A near verbatim transcription of episodes was created, that included additional detail about the files worked on, relationship to other films, and related screenshots. Screenshots were taken to clarify what was said at points in a handling sequence and also to track shifts in activity in the software environment. Transcripts were annotated to reference screen grabs and relevant action.

Content logs were not maintained for episodes 11-20; however potential incidents were highlighted in-line immediately upon finishing the transcription. The catalogue developed for the first ten episodes was refined and extended to include information for all twenty episodes; this information included file names and notes to related content.

Exclusions: The audio track for episode 19 was not recorded at the same time as the screen-cast, making it impossible to analyse activity at a sufficient level of detail. Episode 20 was recorded after a several week-long break, and took a different format to those previously recorded, adding a superimposed video of the developers over the screen cast. These episodes also mark the introduction of a new development environment; work is performed on a new laptop running a different operating system.

Beyond Episode 20: Video for episodes 21-50 was sampled to determine visual and audio quality, to gain a sense for the content of the episode, and to roughly catalogue files that were touched. Notes were added to the master catalog about content when striking evidence of a potential incident was observed, or when issues relating to work prior to episode 20 was mentioned. Episodes 26 and 27 have been fully transcribed for future analysis.

C.2. Incident Catalogue

The table below details features of forty-three incidents around which analysis centred. An additional twenty-five incidents that were considered are not reported here. Eleven were used to develop contextual understanding, while fourteen were related to conceptual design or to global aims for the project. Though they have been used to inform analysis, their data does not cohere with the incidents catalogued below. For example, in the case of contextual issues, the issue may have involved problem-solving, but not clear stages of handling. In particular, recovery may not have resulted in changes that were made or identified within a particular tool or file; the resolution may instead have come in the form of satisfaction or consensus about an idea.

Entire rows have been shaded to indicate issues that were deferred (light yellow) and issues for which problem solving was aborted on film (light orange) . Cells in the Cue column have been coloured (light green) to indicate action based detection. The duration of handling has been marked in **red and made bold** to indicate incidents longer than five minutes in length.

- **Episode number (Ep.)** The number corresponds to numbering on internet hosting site, the letter is the identifier assigned to an incident during analysis.

- **Time** indicates the start and end points of incidents. The start time for incidents were marked at the point at which a task related to the initial detection was ascertained to have begun. The duration of incidents that took longer than five minutes to handle are marked in a **red, bold font.**
- **Description (Driver)**, a brief characterisation of the task. The developer at the keyboard is marked in parenthesis.
- **Detection,** the verbal response given by one or both developers at or near the point of detection. Corresponding indicators of identification or recovery have not been given because those moments are not so clearly defined or relatable to one another.
- **Cue,** what is believed to have spurred detection. In most cases these are outcome-based. Action-based detections are highlighted in green.
- **Location,** the source of the problem based on the outcomes of the identification and recovery.

Ep.	Time	Description (Driver)	Detection	Cue	Location
1-A	08:18-08:52	Marcus shows Joe how to remove strange behaviour in the development web server. (Joe)	M: So we have a problem there, and that's a funniness...	Visual. The webpage; does not properly render.	Tool Behaviour
1-B	11:00-12:00	Joe borrows from an old CSS file to improve layout. (Joe)	J: Oh, I have to remember how to do this.	Action-based, memory related.	Syntax (CSS)
1-C	15:00-17:40	Marcus uses incorrect wiki editing syntax to define a variable. (Marcus)	J: Do you need a space? Before the first curly. M: What have I done wrong?	Joe:Textual, preemptive while a variable is typed. Marcus: Textual, system response in the wiki "undefined variable".	Syntax (Wiki)
1-D	20:00-20:41	Marcus uses incorrect wiki editing syntax, resulting in a rendering error.	J: Oh it's not the first character. M: How did I do that?	Visual. Text added to a page doesn't render properly when saved.	Syntax (Wiki)
2-A	09:02-11:29	Joe questions the addition of whitespace characters. (Marcus)	J: Okay. Do we need the r n r n r?	Visual. Upon seeing how the web page renders.	Syntax (Wiki)
2-B	12:49-14:18	Marcus can't remember a package name. (Marcus)	M: I think its, isn't it? J: I don't know is it, for the...?	Action-based, while adding a package name to a wiki page.	Info. Arch. (Package Structure)

2-C	20:43-23:03	An acceptance test isn't running in the wiki. (Marcus)	M: Cool, right. + Now. J: Ugh	Textual. Upon seeing message in acceptance test. Cursor moves along a message on the output of running a test.	Config. Test runner
3-A	5:39-7:39	An acceptance test isn't running in the wiki. (Marcus)	J: --Ooh. M: Why did that work? J: No it didn't work, you've got that exception. M: Ah the--	Joe: Visual, notices "Output Captured' warning graphic in browser. Marcus:Aural, when Joe points the message out.	Tool Behavior (Wiki Server)
4-A	04:50-06:00	Marcus can't remember wiki link syntax. (Marcus)	M: Ugh, I can never remember which way around--	Action-based.	Syntax (Wiki)
4-B	06:00-07:05	Marcus can't remember how he has organised wiki pages. (Marcus)	J: Ugh, what's that complaining about? ... that looks all right to me. M: No it's not.	Visual, A link created in a wiki page appears as a yet to be created.	Info. Arch. (Wiki)
4-C	09:04-09:49	Joe and Marcus don't like the rendering of a wiki page. (Marcus)	J: Why is child pages centred like that?	Visual. Web page rendering.	Design (UI)
4-D	09:49-11:00	Adding a hardrule to a wiki page. (Marcus)	M: No. J: No, other way.	Action-based, while adding the hardrule.	Design (UI)
4-E	18:00-19:13	Test fails, class not found. (Joe)	J: Oh that's interesting. ++	Textual, error message in problems pane.	Config. (IDE build path)
4-F	19:52-22:11	Defining behaviour in a test, selecting between Concept1 and Concept2 (Joe)	J: [Narrating] Should ask [Concept1] to establish context-- M: --No ! '[Concept2]'.	Action-based, caught by Marcus as Joe narrates the words he is typing.	Design (Object Model)
6-A	01:57-02:47	Distinguishing between instances of Concept 1 and Concept 2 in a test (Marcus)	J: It should find an [Concept1] for a different [Concept2]	Action-based, caught by Joe as Marcus narrates the name he is giving to a test.	Design (Object Model)I
7-A	02:57-03:59	Incorrect class declaration. (Joe)	J: Why is that complaining?	Visual, red bar in the IDE. Cursor hovers over the red bar, revealing message. This action is performed twice.	Implementation

7-B	**11:13-22:49**	Unexpected error messages in tests related to capitalisation of class names. (Not Clear)	J: And we expect this to fail. It is going to say, I can't find a dummy [Concept2] with a small 'D'. Oh- and it has, look, its got a little red thingy saying NoClassDefFoundError. That's funny, I thought it was supposed to say, uh shouldn't that come back as...	Textual, error message in the problems pane.	Design (Global Aim)
8-A	05:30-08:25	Null pointer exceptions - classpath issues - IDE memory caching (Marcus)	J: Nooo. [Clicks 'Output Capture' link, a stack trace appears] M: Excellent, what did we do wrong?	J: Visual, "Output Captured' warning graphic in browser.	Tool Behaviour (IDE)
8-B	10:54-14:41	ClassNotFound Exception reported in the wiki. (Marcus)	M: It wasn't able to find a role.	Textual, stack trace displayed in failing test.	Config. (Wiki CP)
8-C	19:26-24:03	Concept confusion, difficulty using JAVA Generics. (Joe)	M: With an import? J: No it's done that. Ah, it's saying dummy isn't performable as a. Uh.	Visual, red bar in the IDE after a return statement is written in a method.	Implement / Language(JAVA)
9-A	02:26-04:22	Repurposing a method with IDE command results in a duplicate method. (Marcus)	J: Why is that red at the moment?	Visual, red bar in the IDE. Driver moves cursor to red bar, revealing message.	Implement
9-B	15:33-19:20	Refactoring ->Extract Class command within the IDE fails (Marcus)	M: Oh, why has it not worked?	Visual, the IDE does nothing when the command is entered.	Tool Behaviour (IDE)
10-A	08:22-11:51	A broken test is reported as passing. (Marcus)	M: Oh, that's interesting. It's passed! J: It can't have. M: It can't have.	Textual, output from the test runner in the problems pane.	Tool Behavior (Test Runner)
10-B	20:12-22:50	Watcher points out flaw in algorithm. (Joe)	M: Oooh! J: Ugh, that's interesting.	Report: What happens if there are multiple spaces? Followed by... Textual, output from the test runner in the problems pane.	Language (Java)
10-C	**11:51-19:03**	Implementing the CamelCase (Joe)	J: Oh no, it didn't. Ugh, got "say something".	Textual, a unit test is expected to pass, but fails.	Implement Language (JAVA)
10-D	24:29-27:05	A message thrown to an exception does not include all of the expected information. (Marcus)	M: Where's the "caused by?"	Textual, error message doesn't contain information that Marcus expects to see.	Implementation

11-A	00:41-02:38	Decision to create a class is immediately re-taken (Marcus)	M: Actually, no!… J: Class Cast Exception. M: I think that was the wrong thing to do.	M: Action-based, as Marcus is shifting windows from the IDE to the Wiki. J: Outcome, presumably textual.	Change of Plan
11-B	02:55-4:44	An apostrophe in an argument causes a red bar (Marcus)	M: Oooh.	Visual, a red bar under a statement passed to an argument.	Syntax (JAVA)
11-C	04:44-05:24	Using Right-click -> Try/Catch block in the IDE fails (Marcus)	M: What the? Why? Joe: Uggoh.	Visual, the IDE does nothing.	Tool Behaviour (IDE)
11-D	16:53-20:21	Incorrect class declaration. (Joe)	J: Oh, that's 'cause it doesn't extend runtime. I was lazy and I didn't (inaudible)	Visual, red bar in the IDE.	Implement
11-E	20:21-22:24	The pair can't remember if they created a class required for a test. (Marcus)	J: Cool, but we're still getting a NotAnAction-Exception, we're still not getting the, oh we still can't find it! So we've implemented all this stuff - oh okay. Why is that not working then?	Textual, reading message returned by test runner in problems pane, memory related.	Implement
11-F	22:22-23:03	Marcus called the wrong method.(Marcus)	J: We shouldn't get that classCast – [Reading the message given in the stack trace] M: Oops J: That's interesting.	Textual, stack trace displayed in failing test on the wiki.	Language (JAVA)
11-G	23:03-25:00	Client-side HTML rendering issue with brackets (<) (Marcus)	J: Something, some role.	Textual, stack trace displayed in failing test on the wiki. Cursor highlights a portion of the stack trace	Syntax (JAVA/HTML)
12-A	14:54-17:05	Marcus suggests that a class be extended. The method call Marcus suggests to use is private, Joe sorts it out. (Joe)	M: Ooh, is this test broken? (a latent detection, comes at the end of Joe's problem solving process.	Textual, checking the method implementation in the class.	Implement (API)
13-A	**04:55-11:59**	Refactoring surfaces the generics issue (Marcus)	M:Ummm. Now that isn't necessarily a mock. Umm, playing.	Visual, red bar, but also possibly aural, while narrating.	Language Design Implement
13-B	**21:26-39:28**	Refactoring a method surfaces the generics issue. (Joe)	M: Why doesn't it like this?	Visual, a red bar.	Language Design Implement
14-A	13:04-16:14	Marcus realizes that a class is too specific. (Joe)	M: ...and in this case, umm, this is why it just doesn't feel right that this is, it's just too specific.…	Verbal, action-based, caught when Marcus explains behaviour to a Watcher.	Design/ Implement
14-B	18:04-20:37	Null Pointer Exception in a test points out problems in an implementation. (Marcus)	M: Oooh J: Oooh M: That's because, we haven't given our--	Textual, error message in the problems pane.	Implement / API

14-C	**21:35-32:14**	An acceptance test is unable to launch a wiki web server. (Marcus)	M: Oooh.	Textual. A Firefox window reports a page load error.	Config (Wiki)
15-A	**00:00-39:52**	Continuation of 14-C	--	--	--
17-A	**03:32-26:33**	A new installation of test runner hijacks episode. (Marcus)	J: Oh, what's happened there. M:...It's a bit hard to do that while this is running. (Sigh)	Visual,tests that are running on an other section of code launch web browser windows.	Tool Behaviour (Test runner)
18-A	05:08-07:41	Spurious error reported by the test runner.(Joe)	J: That's interesting. Element not found exception. That's umm something new. Why is that not working? Ummm.	Textual, message returned by test runner in the problems pane.	Tool Behaviour (Test runner)
18-B	**15:26-27:06**	Sequencing error causes multiple null pointer exceptions (Marcus).	J: Now that's interesting, that we got a whole bunch of null pointer exceptions.	Textual message in problems pane returned by test runner.	Tool Behaviour (Test runner)

Table C.1: Incidents analysed at the desk.

C.3. Incident Exchanges

This appendix includes the full exchanges for incidents presented in Chapter 6, Section 7.3. The headings are topical. Metadata is also provided indicates the episode and timestamp or that corresponds to entries in the catalogue given in Appendix C2, above. Cross references are also given to sections of Chapters 6 and 8 that discuss the incidents.

C.3.1. Slips of Action

This section gives two examples of slips of action, described in Chapter 6, Section 6.3.1 and within Chapter 8, Section 8.1

An example of a slip of action, drawn from Episode 7, 00:06:51.

> *[Joe creates a local variable within a try block, which he tries to reference in another block. This results in a red bar.]*
>
> *Joe (D): No can't do that cause it's there, oh we can move it outside the...*

[He moves the variable outside of the try-catch block, which fixes the error.]

A second example of a slip of action, drawn from Episode 12, 00:04:45.

[Marcus selects a method that has been suggested by the IDE]

Marcus (D): Oops, that's not what I want to do.

[He backtracks and selects a second method]

C.3.2. Prior Experience

In the exchange given below, discussed in Chapter 6, Section 6.3.2, and catalogued as incident 2-C, Marcus recognises having seen and solved a problem that is causing an acceptance test to fail to run in the wiki. He examines prior work to find the solution. The recovery is made by copying and pasting information found in a configuration file into the failing acceptance test wiki page.

00:20:43

Marcus (D): Cool, right. +

Joe (N): Ugh

Marcus: Now this is something to do, I had to solve this recently and I can't remember how I did it.

Joe: It's an import, you need to import it, don't you? Or it needs to be umm, oh wait, it's trying to execute that as a--

Marcus: --It's the, the look. There's a, I did this before. It's to do with the way it does the test running stuff. Let's just have a quick look [Driver opens Eclipse] in examples that we were messing about with hums.

Joe: It would be in the content here, wouldn't it? No (inaudible) ++++ Hmm.

Marcus: That's the one I wanted.

00:23:03

C.3.3. Blame and Severity

In this exchange, discussed in Chapter 6, Section 6.3.5 and catalogued as incident 3-A, in spite of their stated desire to depict development "warts and all", the developers abort filming in this case, and complete the problem solving between recordings.

00:05:39

Joe (N): It's probably because Fitnesse isn't running. No. What's going on there?

Marcus (D): That's nothing to do with us. It worked a minute ago.

Joe: Yeah, it worked on my machine. (Laughs)/(Laughs)/

Marcus: Oh that was the wrong page, wasn't it? I wonder if it's like, no? This is actual- what's changed? What have you done?

Joe: I haven't changed anything [Voce: falsetto]!

Marcus: Look I'm just going to stop...Where is it? Here it is. Okay, let's just stop that. [Stops the web server from within the IDE] Good [Upon verifying in the browser that pages are no longer being served]. Right stopped and we should be able to just kick it off again.

Joe: I wonder if that install story did something dodgy.

Marcus: [Driver restarts server] It is feasible, but I don't think so.

[Page reloaded; exception still being thrown]

Joe: Ugh.

Marcus: Okay, so. ++ This is annoying

Joe: Well you know, warts and all.

00:07:39

C.3.4. Forming Rules-of-Thumb

This section provides the full exchanges for incidents described in Chapter 6, Section 6.3.6.

A Rule-of-Thumb

In the first exchange, catalogued as Incident 1-A, Marcus has seen the issue before, and provides Joe with the steps to work around the problem. The steps are sufficient to advance work, but Marcus does not explain what the changes do. The reflective language used by both developers suggest they do not completely understand why the mechanisms work.

[Joe (D) loads a web page]

00:08:18

Marcus: So we have a problem there, and that's a funniness with FitNesse, that I've noticed happens sometimes. If you actually stop it, now go back to Eclipse and stop it [Only one stop action is performed]. And then start it again ++ Yeah some weird thing it will install all of the files properly and then refresh that page and it will be hunky dory fine--

Joe:--Oh (right)! That is /(Laughs)/ very strange.

Marcus: Very strange behaviour indeed. /Okay/

00:08:52

The following three examples are of a different error. They demonstrate how rules-of-thumb form over time.

The First Occurrence

The first occurrence is catalogued as incident 7-A.

00:02:57

Joe (D): ...why is that complaining?

[Joe highlights a red bar in the IDE, revealing a message in a tooltip]

Joe: Oh that's because we haven't got the constructors.

Marcus (N): That's right.

Joe: Oh, no, that's not, it says it's not a subtype of Exception [He opens the class giving the error]. Oh--

Marcus: --'Cause it doesn't extend RuntimeException /Okay/

[Joe alters the declaration]

00:03:59

The Second Occurrence

The second occurrence, is catalogued as incident 11-D.

00:16:53

Joe (N): Oh, that's 'cause it doesn't extend runtime. I was lazy and I didn't (inaudible).

Marcus (D): But do you know what? /That's fine/ Actually, I think this is the right t--

Joe: Extend-zzz [Suggesting a correction to spelling: "extends", not "extend"]

Marcus: Duh. /Cool/ I think now is the right time to actually put that in there /Yeah/ To be honest.

Joe: So has he got any warnings other than that? No.

Marcus: No.

00:20:21

The Third Occurrence

This is the final occurrence of the issue. Note that it is embedded within incident 18-B at approximately 00:15:26

00:15:26

Joe: Ahh [A red bar has appeared underneath the entire throw statement] So we didn't include the, when we created it we haven't made it extend exception. So now to make it... runtime exception. And we need a constructor with a message…

00:17:13

C.3.5. Error-Directed Practice, Local Problem Solving

The following exchange illustrates local problem solving undertaken in the course of error handling. Error-directed practice is described in more detail in Chapter 6, Section 6.3.3, local problem solving within Sections 6.3.4, 6. 3.5 and 6.5 and within Chapter 8, Section 8.1.

The exchange given below, catalogued as Incident 8-C, marks the beginning of a four-and-a-half-minute incident in which Joe is implementing a method that makes use of JAVA generics. Joe admits at the outset of his task to implement a method that he does not know exactly what needs to be done, and that he is taking an approach he calls ***fake it till you make it.*** He appears to spur the error handling process by typing a return statement in a method, which immediately throws up a red bar in the IDE. Marcus' comment "With an import?" marks the beginning of the identification process. He has taken a guess about what might be wrong in the file.

The references Marcus makes to prior implementations (line 18) mark a shift in this incident from information gathering directed by trying things to the tactical examination of a prior implementation that is known to work. The pair examine the class WebUser and two other files that were previously written by Marcus that use JAVA generics to reference classes. Using information in these files, they are able to piece together the syntax to use.

	Action	Code	Dialogue
01		guess	Marcus: With an import?
02		disprove	Joe: No it's done that. Ah, it's saying dummy isn't performable as a. /Uuh./ Uh.
03		blame	Marcus: So you, why is it? Oh, cause I (sigh)--
04	Joe opens the class that is throw-ing the error.	mechanism	
05		absolve	Joe: What? No that's okay.
06	Joe extends the class	mechanism	
07			Marcus: Yeah that's right, so get it to compile.
08		system re-sponse	red bar under the name of the class that is extended.
09		identify	Marcus: I think, no go on. I think it's an interface, dude.

10	Joe alters the class signature to implement an interface that makes use of JAVA Generics to reference Concept1	mechanism	
11		system response	red bar under the reference to Concept 1
12		detect	Joe: Ah. And then it's a--
13		identify	Marcus: --that's [Concept1] that's fine. /Okay/ Oh no, we have to say what [Concept1]. So we can just make it uh. ++
14		guess	Joe: Can we do a question mark there?
15	[Replaces a reference to Concept1 with <?>]	mechanism	
16		system response	A class not found message returned, the question mark is reverted to reference Concept1
17		disprove	Joe:No.
18		tactic	Marcus: Make it um, well we've got a Web User in the [prior implementation] examples, we could just use that.
19	Replaces Concept1 with WebUser	mechanism	
20		system response	red bar under the reference to WebUser
21	Reverts to <?>		
22		detect	Joe: Hmm.
23		identify	Watcher: I think you just need to implement performableAsA .
24		affirm	Joe: Really, okay. Didn't we do that? Yeah, it's there. Uh.
25		tactic	Marcus: Would you like to look at another action?

C.4. Sources of Data

Data for the project was collected from sources created in 2009 that were published and accessed on the internet.

Video Recordings

Sixty videos were uploaded to a video hosting site between March and August 2009. The first was recorded on 14 March, 2009; it was not uploaded until 24 March, 2009. The last video was uploaded on 4 August, 2009. Forty-nine videos were assigned an episode

number by the creators, given a title, and a description highlighting aspects of the content. The final ten (Ep. 50-60) were assigned episode numbers, but were not given a title or description. These were uploaded on two dates, 13 July, 2009 and 4 August, 2009. Commit information from the corresponding hosted code repository suggest that the work performed in the latter episodes is maintenance to existing functionality.

The videos comprise thirty-one hours, fifty minutes. Videos range in length from 07:42 (Ep. 3) to 53:17 (Ep. 60). The average length is 32:22. Some episodes are recorded on the same day, on other days, only one episode is filmed. The metadata on the video hosting site indicates that videos were uploaded in batches, and the developers do not consistently indicate the date on which they are recording. In latter recordings (Ep. 20 and beyond) work performed in the same coding session can be established by examining the clothes that the programmers are wearing, which are visible in a video of the programmers that is super-imposed over the screen-cast in the lower right-hand corner of the film. In earlier videos, this information was reconstructed using commit information in the source code repository, and by examining posts made to Twitter and Facebook.

Damage: Episodes 21 through 34 suffer from technical problems. In four (Ep.21-24), an audio echo is present that obscures the dialogue, while in episodes 25 through 30 there is a latency between the audio, screen-cast and video of programmers, sometimes as great as fifty seconds. In another four episodes (Ep. 31-34) either audio, video or both are lost during the recording. This damage prohibits detailed analysis of the relationship between what the developers say and what they do.

Gaps in time: Filmed episodes are separated by breaks in time. Sometimes the interval is as short as a few minutes, at other times days or weeks. There are several mentions of work done without recording, or of promises to return after a short break when in fact filming ceases for that date. One episode (Ep. 19) has audio commentary recorded over the video after a 3-4 week break.

Format, Episodes 1-18: The developers introduce themselves and usually announce the episode number. Occasionally, they give the date on which the recording was made. The developers also introduce others who are watching the taping. There are several instances in which the developers use the first couple of minutes to explain what happened in a previous episode, or to provide background for a particular choice. In general, the pair aim to program for one 25-minute session. The pair are working on a Windows laptop, using Firefox and the Eclipse IDE, and the background noises suggest that the work is being done in an office.

Episode 19: The audio for this episode was recorded over the screen-cast, several weeks after creation. The developers reflect on what happened in this episode, as well as on their overall work practice for the project. One of the things that comes out of this is a solidification of commit strategy in the version control system, both in terms of the information that should be included in messages, and in frequency. This strategy is apparent after Ep. 19, when the overall number of commits increases, and is performed on-screen, rather than during breaks. In addition, the information given in commit messages is better structured and includes references to the episode number in which it was performed. This episode is the first in which the pair are working on a Mac laptop.

Episode 20 and beyond: This episode marks several shifts in the recording and working environment. From this point onward, the videos include a small video of the developers superimposed over the lower right-hand screen of the screen-cast. In addition, work appears to be performed in a home environment. Though the developers announce in Ep. 19 that they will no longer web-cast the episodes, there is some evidence that they continue to use web meeting software.

Social Media

The developers used Twitter and Facebook to inform followers of project activity; both sources were used to corroborate dates for early programming sessions. The Facebook page also included photos of recording equipment, and of one office in which work in early sessions was performed. The photos corroborate the understanding that early sessions were office-based, as do posts in which the developers ask followers for office space to use for filming.

Blogs and Websites

The project has a website, with links to videos and source code. It also contains general information about the project. At one point the site also included notices of upcoming events. The site went offline in March 2012, but was brought back online in a slightly altered format at the end of 2012. Both developers have personal blogs. These were referred to for background information.

Source Code Repository

The software created in the project is hosted on a publicly hosted code repository. The metadata in this repository was used to corroborate dates for programming sessions.

D. Notes on After the Fact

The study *After the Fact* is reported in Chapter 7. It drew upon data collected at Site B and Site D. For an overview of sites, see Chapter 4, Section 4.3. Other detail about data collection and analysis are reported in Chapter 4, Section 4.4.3, and in Chapter 7, Section 7.2.

D.1. Transcription and Field notes

A near-verbatim transcription was created of six interviews gathered at Site B, and three interviews gathered at Site D following the conventions given in appendix A.1. At both sites, analysis began at the point of collection. Terms used were checked and information previously given was stated back at multiple points for clarification and correction. In several instances, restating information to informants resulted in the addition of omitted details.

Immediately following each interview, notes taken during the interview were annotated and expanded. In addition, reflection was made to describe impressions and details of the major topics raised in the interview, and to evaluate application of the method. Informants were sent follow-up email messages seeking additional materials mentioned during conversation, and they were invited to provide additional comment. Informants from Site B have also been sent draft copies of reports featuring their account.

D.2. Critical Decision Method Protocol

This section summarises protocol suggested for conducting a CDM interview as described in *Working minds: A practitioner's guide to cognitive task analysis* (Crandall, Klein, & Hoffman, 2006). Additional notes explain how the protocol was adapted and applied in interviews collected at Site B and Site D.

Critical decision method interviews are typically conducted by two researchers working together. One interviewer asks questions of informants, while the second researcher takes notes about responses. Impressions of the interview are shared between researchers immediately afterward. Interviews are generally also audio-recorded, and transcribed for analysis.

The CDM interview protocol is semi-structured and flexible. The aim is to collect a rich story using questions to probe for more detailed information about events and cognitive phenomena. The emphasis in incident identification is to examine novel or unusual problems that a participant has encountered on the basis that this will reveal more detail about how problem solving or decision making is performed on the job. The protocol entails examining a single incident in four semi-structured ”sweeps” that establish features of critical decision making from different perspectives.

In the study reported in Chapter 7 interviews were conducted by a single person, as it was difficult to arrange paired interviewing. The interviewing process can take several hours, which was also deemed to be impractical. The managers who granted access at Site B offered free access to approach and arrange meetings with developers and were informed that sessions might span two hours speaking. Managers at Site D granted access for an hour, and permitted developers to “cost” an hour of working time to the interview.

Interviews were arranged in person or by email, and each person was sent an information sheet before the appointment (see also appendix D.4). The information sheet was reviewed with the informant before the conversation, and each person signed an informed consent form. Interviews were audio-recorded, and notes were taken. Interviews concluded with questions about background: time spent in the organisation or on a team, time spent professionally making software, and details of education and training.

The aims of the study were not to establish error in relation to education or experience, so a soft touch was taken in collecting these details. This proved to be a useful tactic at Site B, in which all of the informants were very well educated and/or experienced, but had not necessarily taken qualifications related to computing. At both sites, the approach was noted to have the effect of communicating to informants that judgements were not being made about the story they had given and their level of expertise.

Sweep One: Incident Identification

In the first sweep, the participant and the researcher identify a critical incident, and the participant gives a brief account of what happened. Crandall, Klein and Hoffman identify four elements of selecting a useful incident (2006, adapted from pp. 7-76):

1. **Relevance:** The person must recount a story in which they were a “doer” or decision maker. Witnessing an event is not the same as actively participating. Did a person’s actions have a direct impact on the outcome of the event? Given an outline of the

event, it should be apparent to the interviewer if it meets criteria for the area of analysis.

2. **Character:** Asking a participant for a strange or "weird" incident may result in interesting stories, but may not result in stories that can be analysed for evidence of decision making or other cognitive activity. What constitutes "critical" in a domain may not be initially apparent.
3. **Listening and Prompting:** Telling participants before meeting them that you are seeking a story may cause them to select, omit, refine and rehearse details of the story that will suppress information relevant to the research. At the point of interview, it is important to allow participants to identify a story of interest, and to listen and wait as they recount details. It may also be necessary to prompt them to carry on if focus is lost, or if they are not sure about the kind of information the interviewer is seeking.
4. **Structure:** The participant provides the structure of the interview, through the content of the story and the details they provide about sequence, beginning and end points. Incidents may begin earlier than the point established by the participant, and they may have alternative endings.

In this sweep, each informant was asked to think of an incident from recent work that was challenging or that had been particularly difficult. Table D.2 provides examples of suggested prompts followed by questions that were asked during this phase. In some cases, more than one possible incident was reviewed. In one case the researcher suggested an incident; in another case, the informant selected to recount an incident he felt was more relevant. In order to improve precision of recall, and to hear stories that were "fresh", informants were asked to recount an issue encountered in recent work, defined as work that had been done in the past week or two. This adaptation is in-line with other documented adaptations to CDM that seek incidents in the "here and now".

Suggested Prompts:	- Can you think of a time when you and your skills were really challenged? - Tell me about the last time you… - Can you think of a time when your skills really made a difference? - Maybe things would have gone differently if you were not there?
Questions Asked (edited for clarity)	**Notes**
Among the projects you have described, can you identify something you've worked on in the last two weeks that was challenging for you?	This example demonstrates using a warm up to identify possible projects.
You mentioned the other day that you have been doing data modeling for the ___ project /Yes. / and I'd like to talk about that I think… I'd like to see if the incident you described to me the other day might be worth pursuing in this conversation. So you mentioned that you started thinking about using an XML model on that project /mm hmm/ but you decided to stay with a database, a relational database. Do I remember that correctly? /Yes. /	In this project, the developer mentioned a possible incident when the interview was arranged.

Okay, so this can be a big problem, it can be a small problem it can be something that you were tearing your hair out about or that just took a few minutes but that you remember and sort of made you stop for some amount of time in your work.	This was the last interview, possibly the most useful prompt.
[W]hat we need to do is find some recent problem that you've been working on. And it can be from this project, it is always nice to start from something that is fresh in your mind or it can be another project.	Recent work is indicated as preferred.

Table D.2: Prompts for incident selection *After the Fact.*

Sweep Two: Timeline and Decision Point

In the second sweep, a timeline is established to note critical decision points. A critical point is one in which the participant experiences a major shift in thinking or understanding about a situation, or takes decisive action. They are critical in the sense that they are "turning points" at which different decisions or actions may have been taken (Crandall et al, p. 76).

Establishing a timeline requires determining a scale that is appropriate to the incident. Some incidents involve specific timings and durations that are important to understanding what went on. Other incidents may involve elements that are temporally distant from one another. In the case of the latter, it may be sufficient to note the sequence of events and their relation to one another over time.

The sweeps are described in the guidelines as unfolding more or less sequentially, with specific time devoted to plotting the timeline on a whiteboard or paper that can then be used in subsequent sweeps. It emerged in practice that it was more natural to allow the conversation about particular details of the incident to unfold and to periodically establish the relation of events to one another in time. Rough timelines were sketched in the field book, and details were checked with informants. Probes were used to establish how one decision or action related temporally to others.

Sweep Three: Deepening Probes

The process of establishing a timeline interleaves with a more detailed recounting of the incident itself. In the process, deepening probes are used to elicit information about cues and patterns the participant perceived, the rules-of-thumb they devised, the kinds of decisions they had to make, and details about particular cases. The critical decision method is often used with a small set of deepening probes to examine one or two cognitive phenomena, such as the information or guidance that is sought and used.

Probes fall into four broad categories (Crandall, Klein, & Hoffman, 2006, p 80). If the critical point in the process involved:

- ***Observation***, then probe for information and cues. Seeking guidance from others also falls into this rubric.
- ***Making sense of a situation***, then probe for assessment and mental models. Analogues might also serve.
- ***Decision making***, probe for decisions, goals and objectives.
- ***Knowledge***, use probes about experience, and options. Establishing whether or not the case was standard would also be helpful.

In the interviews collected at Site B and Site D, opportunistic use was made of probes from all categories, on the basis that all of them might yield useful information about an error. Deepening information also emerged from subtle probing about time, such as by asking an informant to recount what happened next, or asking how they understood what to do next.

Sweep Four: Hypothetical Alternatives

Finally, each participant is asked to consider hypothetical alternatives to decisions that were taken, or to consider how someone else might have handled an incident.

In the study reported in Chapter 7, hypothetical alternatives were volunteered in several interviews. One account included constraints on problem-solving imposed by organisational practice. In this case, a set of alternative circumstances that would have avoided the error or would have eased recovery were clear. However, in general responses to this line of questioning were sceptical, or dismissive. Informants indicated that there were not other things that could have been done, or noted that alternatives (such as greater knowledge) might have helped, but were impractical.

D.3. Coding

Nine transcribed interviews were read and annotated to identify themes in the data. The designation of sweeps to gather different kinds of information in Critical Decision Method interviews provided a structure for grouping data during analysis (for a fuller description of the protocol, see the prior section). Information related to selection of incidents was identifiable, as was information given in response to deepening questions. Transcripts were first coded into segments. Segments were identified by questions and responses that moved discussion in a distinct direction; this determination was made by assessing how an area of the transcript broadly corresponded to targets for the different sweeps of the interview:

Identification and Accounts - this was used to segment the initial identification of incidents, but was also used to encapsulate later complete accountings of the incident by the respondent.

Juncture in Time or Decision Point - used to segment interviewer recapitulations of previously given information, and also to note questions and responses to "what happened next" or "what did you do then" prompts.

Deepening - the suggested prompts for deepening probes did not always correspond directly to questions that were asked or to given responses. Recommended prompts were not always used and responses often voluntarily included detail that could be broadly identified with one of the deepening categories.

Hypothetical Alternatives - though Crandall et al. describe this sweep as roughly following the deepening sweep, this kind of questioning was used at different points in the interview to probe for greater detail as required. It was also used by the interviewer to demonstrate technical knowledge if the sense was given that the informant might be withholding information or tailoring based on their understanding of my expertise.

Each interview was coded into between 30 and 45 segments; segments often included more than one question and response and almost certainly included information relating to more than one category. Multiple categories were often assigned to reflect evidence of more than one area of deepening, such as a response that described information that was sought, and how that information related to goals or priorities.

D.3.1.Codebook

This is the codebook that was developed out of analysis of the interviews collected at sites B and D. The four main numbers correspond to the sweep of the interview. Section 3, Deepening Probes was initially populated using categories of suggested prompts, but were iteratively developed during analysis into a set of terms that reflected the content in the interviews.

1. Identification
2. Juncture in Time or Decision Point
3. Deepening Probes
- ***3.1. Cues***
  - 3.1.1. Talking through
  - 3.1.2. Seeing
  - 3.1.3. Chance
  - 3.1.4. Error
  - 3.1.5. No change
  - 3.1.6. Timeliness
  - 3.1.7. Votes
- ***3.2. Information***
  - 3.2.1. Colleague
  - 3.2.2. Environment
  - 3.2.3. Client
  - 3.2.4. Documentation
  - 3.2.5. Collective
  - 3.2.6. Code
- ***3.3. Analogs***
- ***3.4. Standard Operating Procedures***
  - 3.4.1. Individual
  - 3.4.2. Team
  - 3.4.3. Organisational
- ***3.5. Goals and Priorities***
  - 3.5.1. Individual
  - 3.5.2. Team
  - 3.5.3. Organisational
  - 3.5.4. Commercial
- ***3.6. Options***
- ***3.7. Experience***
- ***3.8. Assessment***
  - 3.8.1. Foresight
  - 3.8.2. Hindsight
  - 3.8.3. of Performance
  - 3.8.4. of Solution
  - 3.8.5. Naming
- ***3.9. Mental Phenomena***
  - 3.9.1. Feeling
  - 3.9.2. Thinking/Imagining
  - 3.9.3. Insight
  - 3.9.4. Memory
  - 3.9.5. Giving Up
  - 3.9.6. Expectation
- ***3.10. Problem Solving***
  - 3.10.1. Decision Making
  - 3.10.2. Explaining
  - 3.10.3. Tactic
  - 3.10.4. Strategy
  - 3.10.5. Diagnosis
  - 3.10.6. Understanding
  - 3.10.7. Learning
  - 3.10.8. Communicating
  - 3.10.9. Reading
  - 3.10.10. Comparing
  - 3.10.11. Drawing
  - 3.10.12. Reasoning
  - 3.10.13. Questioning
  - 3.10.14. Fitting
  - 3.10.15. Delaying
  - 3.10.16. Implementing

3.10.17. Checking
3.11. Guidance
3.12. Side turns
3.13. General Knowledge
3.13.1. Technical
3.13.2. Organisational
3.13.3. Cultural
4. Hypothetical Alternatives
5. Personal Information

D.4. Information Sheets

Two information sheets were used. Each contained the same information about researchers, contact information and a slightly modified version of the expectations for the interview. Modifications were made to the timeframe for the interview to reflect tighter constraints at the second site, and the language used to frame the research was refined. Participants at both sites were informed that was sought about "things that go wrong" in development, though the terms used to describe those things varied slightly.

In the first information sheet (see Figure D.4.1), the research problem was framed in terms of bugs, and the focus of analysis was to explore how developers deal with "small mistakes". The second information sheet does not use the term bugs, and removes the emphasis on personal responsibility conveyed by the term "mistake". The aim for analysis given was to understand how developers manage "problems encountered" in everyday work. The second information sheet (Figure D.4.2) also included a graphic, which was intended to catch the eye of potential participants who were solicited through an invitation sent by email.

Things that Go Wrong in Software Development

What is this research about?

Though bugs in software are well studied, not much is known about their history. In particular, the experience accumulated by developers in dealing with small mistakes is not understood. This research examines how software developers understand, manage and communicate about things that go wrong in their everyday work. It will yield new insights about software bugs, and recommendations for improving software engineering practice.

Who are you?

We are researchers at The Open University, UK.

What do you want from me?

We would like to talk to you about your experience with software development. We are seeking an in-depth understanding and would like to talk with you for up to two hours. However, this can be done over multiple sessions if you prefer. We may also want to observe you while you perform software development activities, and to email you to clarify information you provide to us.

What about my privacy?

All information is stored in encrypted folders. The only people with access to folders are the research team. You can withdraw your data from the study at any time.

No information that personally identifies you will ever be disclosed to anyone outside of the research team.

I have more questions

If you have questions about this study, feel free to contact the principal investigator, Tamara Lopez, via email (t.lopez@open.ac.uk). Alternatively, you can contact the project supervisor, Prof. Marian Petre (m.petre@open.ac.uk).

How can I participate?

Please contact Tamara Lopez (t.lopez@open.ac.uk).

Open University Human Research Ethics Committee Reference No. HREC/2011/#1101/1

Figure D.4.1: Information sheet for Digital Humanities (Site B).

Examining Things that Go Wrong in Software Development

What is this research about?

Software rarely works as intended when it is first written. Things go wrong, and developers are commonly understood to form theories and strategies to deal with them. This research examines how software developers understand, manage and communicate about problems encountered in everyday work. It will yield new insights about how software development expertise accumulates, and recommendations for improving software engineering practice.

Who are you?

We are researchers at The Open University, UK.

What do you want from me?

We would like to talk to you about your experience with software development. We are seeking an in-depth understanding and would like to talk with you for around an hour. However, this can be done over multiple sessions if you prefer. We may also want to observe you while you perform software development activities, and to email you to clarify information you provide to us.

What about my privacy?

All information is stored in encrypted folders. The only people with access to folders are the research team. You can withdraw your data from the study at any time.

No information that personally identifies you will ever be disclosed to anyone outside of the research team.

I have more questions

If you have questions about this study, feel free to contact the principal investigator, Tamara Lopez, via email (t.lopez@open.ac.uk). Alternatively, you can contact the project supervisor, Prof. Marian Petre (m.petre@open.ac.uk).

How can I participate?

Please contact Tamara Lopez (t.lopez@open.ac.uk).

Open University Human Research Ethics Committee Reference No. HREC/2011/#1101/1

Figure D.4.2: Information sheet for Course Planning (Site D).